ON THE RELATION OF REFINEMENT BETWEEN ALGORITHMS

-- Preliminary Report --

Robert L. Constable

TR 73-187

December 1973

Department of Computer Science
Cornell University
Ithaca, New York 14850

ON THE RELATION OF REFINEMENT BETWEEN ALGORITHMS

-- Preliminary Report --

Robert L. Constable

TR 73-187

December 1973

Department of Computer Science
Cornell University
Ithaca, New York 14850

On the Relation of Refinement between Algorithms

-- Preliminary Report --

On the Relation of Refinement between Algorithms

-- Preliminary Report† --

Robert L. Constable
Department of Computer Science
Cornell University
Ithaca, N.Y.

Abstract:

We define a class of tree schemata and a notion of refinement between trees (finite or infinite). These concepts allow us to talk about different "ways to program an algorithm" and the "structure" of an algorithm. We obtain as a special case certain concepts such as "approximation," "convergence" and "fixed point semantics" recently employed by Scott to describe the semantics of flow diagrams. We concentrate in this paper on relating our concepts to Scott's.

† This work was supported in part by National Science Foundation grant GJ 579 .

ON THE RELATION OF REFINEMENT BETWEEN ALGORITHMS

Robert L. Constable

§1. Introduction.

When people talk informally about programming and computing they use many interesting and apparently meaningful concepts which at present have no precise meaning. They say, "This program is easy to understand." "This Algol program and that FORTRAN program are just different ways to code the same algorithm." "This is a top down, structured, way to program." "For this problem we see an apparent trade-off between structural complexity and efficiency."

One of the goals of computing theory is the clarification and organization of these imprecise concepts that appear useful in the conduct of computing. Perhaps some of the concepts can not be made precise, and in discovering this we may learn to modify our intuitive understanding of computation. On the other hand, certain of these concepts can be made precise. In the brief history of computing theory we have given exact meaning to such concepts as, "computable", as in "this function is not computable," or "optimal" as in "this function has no optimal program," or "computational complexity" as in "the computational complexity of this parsing algorithm is asymptotically n^3."

In this paper we describe an attempt to make more exact the concept of an algorithm as distinct from a program and we attempt to give exact meaning to certain uses of the phrase "a way to program this algorithm." The concepts we develop will also allow us to discuss certain aspects of the structural complexity of programs.

Ultimately we hope to develop these concepts to the point where we can discuss the issue of trade-off between structural and computational complexity. Furthermore we feel that these ideas will help in solving problems in computational complexity theory which appear to need new concepts, such as the problem of describing the class of natural complexity measures and feasibly computable functions.

We begin with a notion of program refinement at the level of program schemata. This will allow us to talk about the development of an algorithm by successive refinements. We examine this concept in depth and relate it to the recent work of Dana Scott on the semantics of algorithmic languages.

The connection of this work to Scott's is one of the main topics of this paper. We feel that this more general approach to the semantics of programs makes Scott's work on the lattice of flow diagrams, [16,17,18] clearer and more computationally meaningful.

The mathematics used in this paper is mainly constructive. Some of the concepts have a different meaning in classical mathematics. We will alert the reader to these differences when they arise.

of this concept are not constructively correct. We use the idea that a tree is a connected acyclic directed graph.

Definition 1: To construct a tree one must define an enumerable set N of nodes (points), $\{n_0, n_1, n_2, \ldots\}$ and an enumerable set of pairs of nodes, $\langle n,m\rangle$ called edges, which enter m and leave n and a distinguished node n_0, called the root, such that

(i) given any node n we can determine the number of edges leaving it and we can enumerate them and

(ii) we can find for any node n except the root a unique path from the root to n.

Remark: This implies that there is never an edge $\langle n,n\rangle$ and that exactly one edge enters each node except the root, n_0, which no edge enters.

The set of all trees over the class of nodes N is denoted T(N). Two such trees are identical iff their sets of nodes and edges are identical.

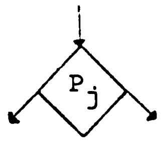
Remarks: (1) A classical definition would not require that we could "find" anything, neither the root, nor the edges leaving a node, nor a path. We will call an object an indefinite tree iff it is a tree except for property (i) .

(2) Because our mathematics is constructive, we are actually able to effectively enumerate nodes and edges.

Definition 2: The depth of a node n is the number of edges in the unique path from n to the root. The out-degree (or simply degree) of a node is the number of edges leaving n. This is either finite or infinite. A node with no edges leaving

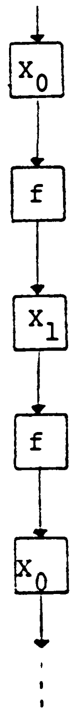
§2. Tree schemata.

Informally a <u>tree scheme</u> is a tree, finite or infinite, whose nodes (with edge attachments indicated) are:

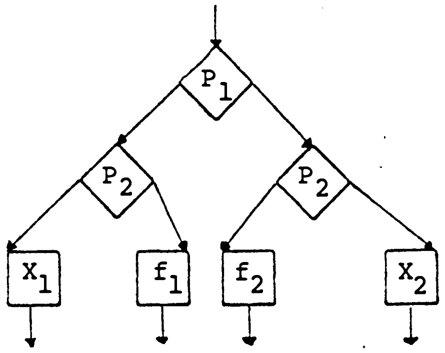


Examples:

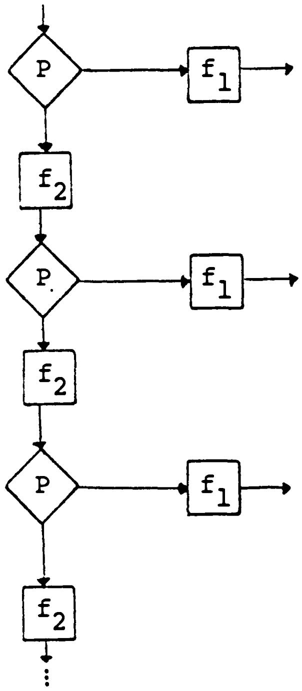
(1)



(2)



(3)



To define these objects precisely we need a precise definition of a tree (finite and infinite). Many classical definitions it is called a leaf. The set of leaves is called the frontier.

The reader can easily verify the following property of trees.

Proposition 1: We can enumerate all nodes of depth n . If the degree of each node is finite, then the number of nodes of depth n is finite.

Remark: A constructive proof of this theorem should allow anyone to calculate the number of nodes of depth n given the finite degree of each node.

Definition 3: A tree scheme is a tree whose only nodes have degree one or two and whose degree one nodes are from the set of function variables, $f_0, f_1, \ldots$ or tree variables $X_0, X_1, X_2, \ldots$ and whose degree two nodes are from the set of predicate variables $p_0, p_1, p_2, \ldots$. The left edge leaving a predicate variable is called the true exit, the right the false exit. The root node is also called the entrance or start node. The leaves are called exits.

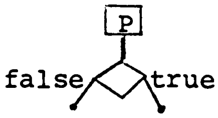
We imagine each function as mapping a state space S to itself

$$f: S \to S$$

and each predicate as mapping S to true or false

$$p: S \to \{\underline{true}, \underline{false}\}$$

To treat predicates with the same generality as functions we should introduce nodes of the form $\Diamond X$. However, this would complicate our treatment of refinement (§ 4). Instead we could treat predicate evaluations similar to function evaluations and provide predicate boxes $\boxed{P}$. Then conditionals would have the form



where we can think of the diamond as testing the result of computing P.

In this paper we adopt a modification of this method. We assume that predicates can be represented by functions $f:D \to D$. This is true as long as D has at least two distinct elements, say $\perp$, T which are distinguishable by some basic predicate. Then $p(x)$ iff $f(x) = T$. Such an approach to predicates results in tree schemes which are identical to those obtained using special predicate boxes, $\boxed{P}$, yet avoids the necessity to distinguish predicate and function boxes. Such an approach is adequate for the results of this report.

<u>Definition</u> 4: <u>To</u> <u>execute</u> a tree scheme t with input functions $f_1,\ldots,f_n,\ldots$ and predicates $p_1,p_2,\ldots,p_n,\ldots$ on state space S,

by tying all leaves together). If t_1 has no exit, then $t_1 = (t_1;t_2)$.

Define $(p \rightarrow t_1,t_2)$ as the tree whose root node is predicate p whose right edge is attached to t_1 and whose left edge is attached to t_2 .

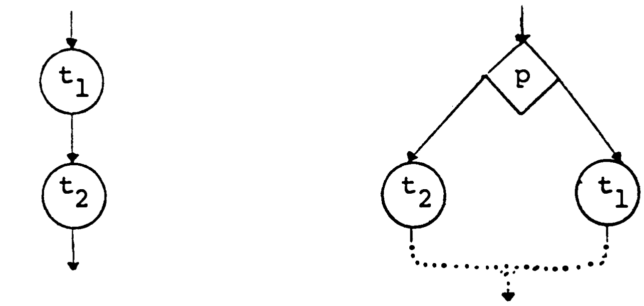
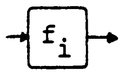
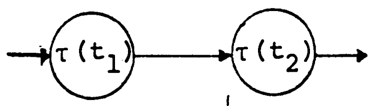
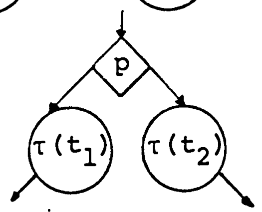


Figure 1.

<u>Definition</u> 2: A (finite) <u>algebraic</u> <u>term</u> is f_i or $(t_1;t_2)$ or $(p \rightarrow t_1,t_2)$ where t_i are terms. The tree defined by term t is denoted $\tau(t)$ and is defined by

(i) $\tau(f_i)$:= 

(ii) $\tau(t_1;t_2)$:= 

(iii) $\tau(p \rightarrow t_1,t_2)$:= 

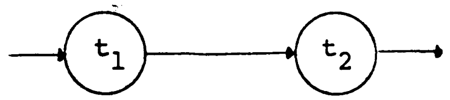
<u>Proposition</u> 1: For any tree $t \in T_f$ there is a finite term r such that $\tau(r) = t$.

Proof: for the reader.

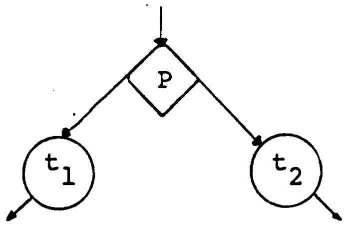
Remark: Given a tree t there may be many terms which describe it. For example, execute the start node on S and then execute the subtree obtained by deleting this node on the state space obtained by applying the operation. This subtree is obtained by deleting the start node and if the start node is a predicate test, then by deleting also the subtree connected to the false exit if the test result is true and to the true exit if the test result is false. If any node encountered is a tree variable, then execution stops (abnormally).

§3. Algebraic structure.

Let T_f denote the set of finite tree schemata and T the set of all tree schemata. We can build finite trees using two simple algebraic operations, the product of trees

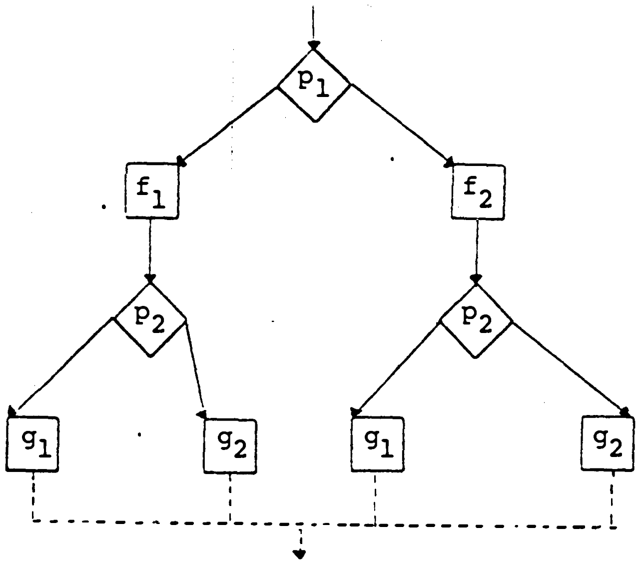


and the sum of trees



These simple operations assume that no tree has more than one exit. To handle more exits, we can either consider more complex operations or we can force trees to have at most one exit. We will do the latter and it is fairly clear that on finite trees we are able to build up the same trees as we could by generalizing the operation.

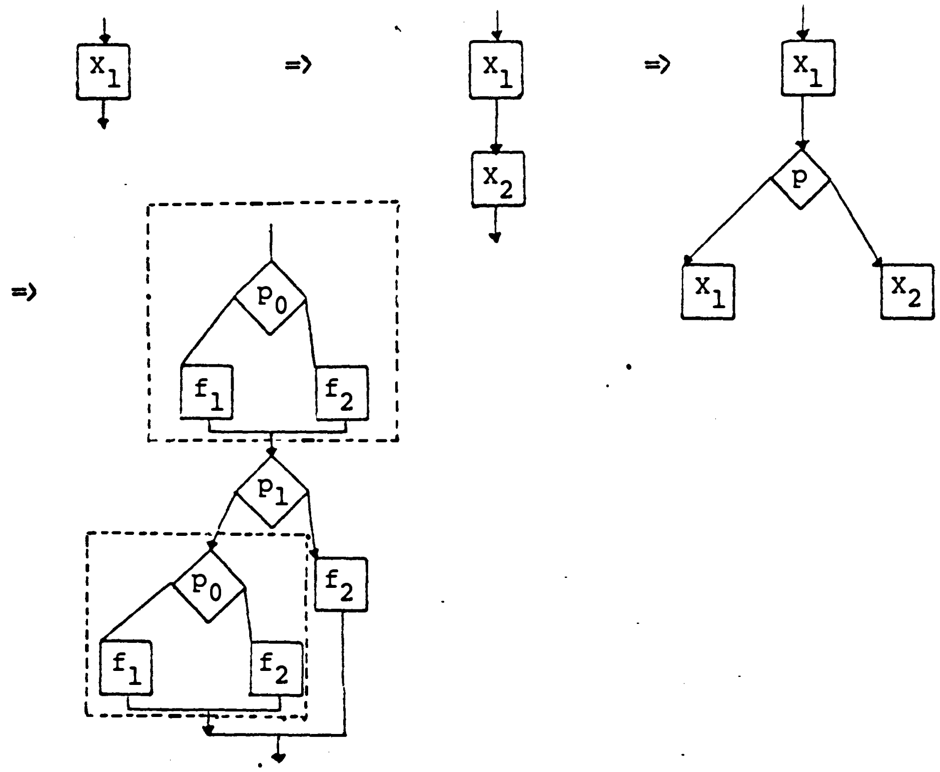
Definition 1: Given $t_1, t_2 \in T$ define the product tree, $(t_1;t_2)$ by attaching t_2 to each exit of t_1. This is equivalent to viewing t_1 as a tree with only one exit (obtained



$$((p_1 \to f_1, f_2);\ (p_2 \to g_1, g_2)) = (p_1 \to (f_1; p_2 \to g_1, g_2),\ (f_2;\ p_2 \to g_1, g_2))$$

§4. Refinement.

The tree variables X_i denote unfinished parts of the algorithm. We can imagine a sequence of trees which represent the way an algorithm is developed. For example,



To express this relationship we want to talk about substituting trees for variables.

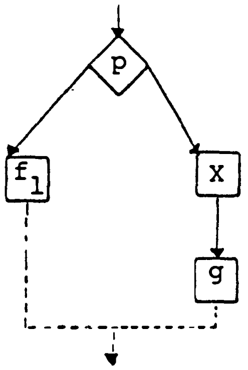
If t is a tree containing the variables $X_{i_1},\ldots,X_{i_n},\ldots$ then we can display this fact by the notation $t[X_{i_1},\ldots,X_{i_n},\ldots]$, where we list only those variables which we want to display.

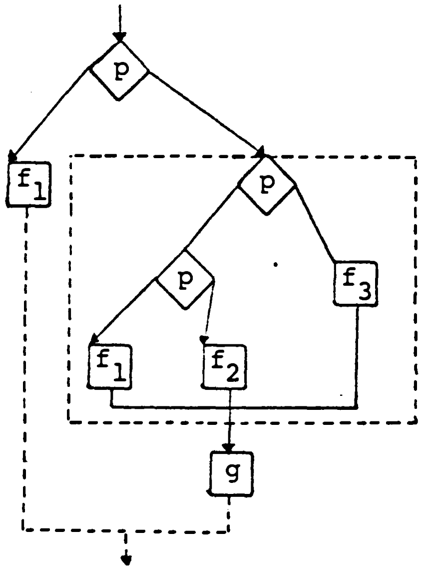
Definition 1: To substitute a tree t for an occurrence of a variable X in tree s, remove the node $\boxed{X}$, attach the incoming edge to the root of t and attach the exit of t (if t has an exit) to the outgoing edge.

Remark: We write the result of substituting t_2 for X in $t_i[X]$ as $t_1[t_2]$.

Examples 1:

(1) substituting a finite tree





or t_2 is obtained by selecting a subset of variables X_i occurring in t_1 and substituting for each occurrence of variable X_i, $i=1,2,\ldots$ a tree t_i which is not a variable and does not have as root some variable X_j, $j \neq i$. When we want a unique variable, $\perp$, we use X_0 as a special variable which can be refined to any X_i, $i \neq 0$.

Remark. (1) The condition that T_i is not a variable and that its root is not some $X_j, j \neq i$ prevents us from simply renaming variables as an act of refinement.

(2) The definition allows us to substitute for infinitely many different variables as one refinement, but for each occurrence of X_i we must substitute the same tree. That is, when we have an algorithm with two occurrences of X_1 we know that these components, whatever they are, must be the same.

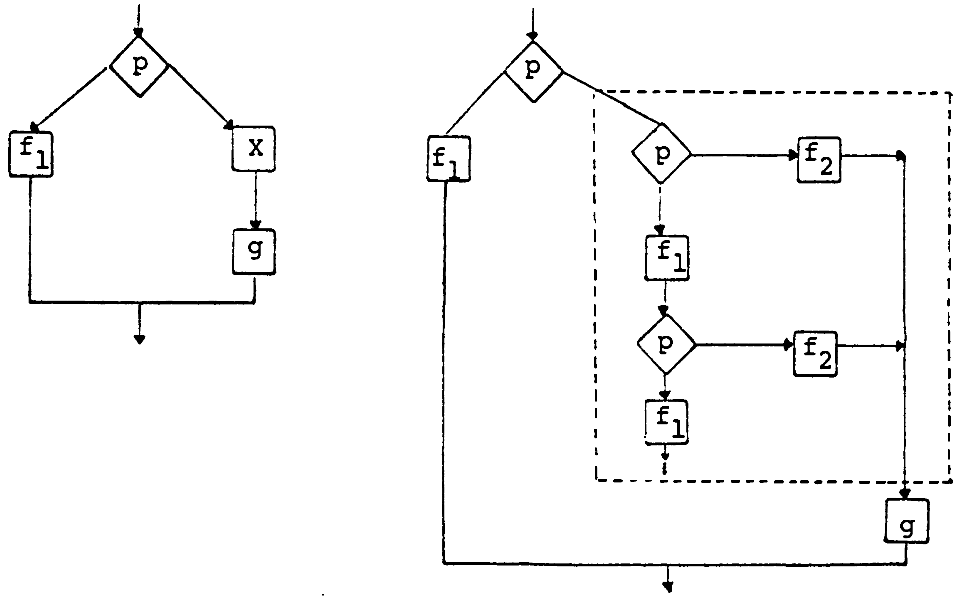
<u>Definition</u> 3: A tree scheme t is <u>structurally free</u> iff no tree variable X_i appears more than once.

<u>Remarks</u>: (1) In free algorithms we have no constraints between undefined parts.

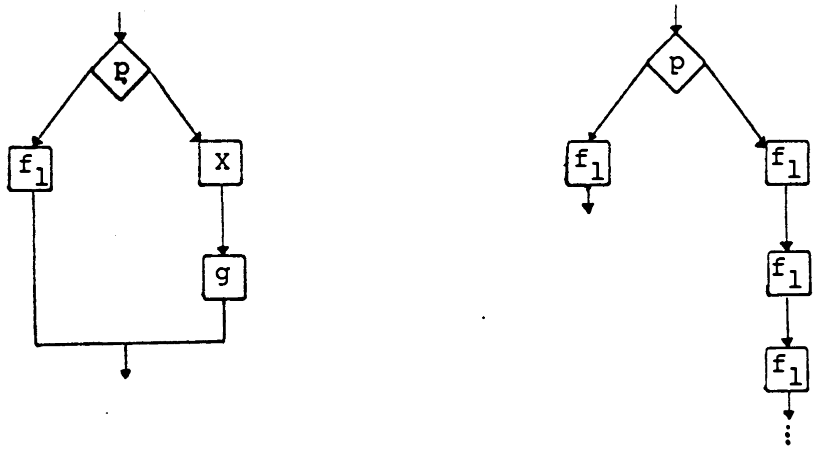
(2) In a structurally free tree scheme, one need not distinguish among undefined components. So a single symbol such as $\perp$ can be used for any variable and we change our notion of substitution so that we can substitute different trees for different $\perp$'s. Scott [16] considers such a notion of substitution.

<u>Proposition</u> 1: The relation $\sqsubseteq$ is reflexive and transitive on T and anti-symmetric on T_f.

(2) substituting an infinite tree



(3) substituting a tree with no exit



We now say how one algorithm can refine another, essentially by "filling in" undefined parts of the other.

<u>Definition</u> 2: Given tree schemes t_1, t_2 say t_1 <u>is refined by</u> t_2 (or t_1 <u>approximates</u> t_2), written $t_1 \sqsubseteq t_2$, iff $t_1 = t_2$

Proof: (1) clearly $t \sqsubseteq t$.

(2) Suppose $t_1 \sqsubseteq t_2$ and $t_2 \sqsubseteq t_3$. We claim $t_1 \sqsubseteq t_3$. Suppose further that t_2 results by substituting s_i for X_i , and t_3 results by substituting q_j for Y_j . In each s_i , substitute q_j for each occurrence of Y_j for all j (there may be no such Y_j). Call the resulting tree r_i (which may equal s_i). Now for each occurrence of Y_j in t_1 substitute q_j . For each occurrence of X_i in t_1 substitute r_i . Call the resulting tree t . Clearly $t_1 \sqsubseteq t$. We now claim $t = t_3$. We show t can be obtained from t_2 exactly as t_3 is. One way to see this is to consider how t_3 is decomposed by the substitutions which formed it.

To see this consider the following shorthand notation: Let t_2 be written as $t_1[s[\ldots y\ldots],Y,Z]$ which means t_2 arises from t_1 by making a number of substitutions of the form $s_i[\ ]$ for X_i and some of these $s_i[\ ]$ may contain Y_i but t_2 also has occurrences of Y_i directly in t_1 along with other variables Z. Then in this shorthand, t_3 is $t_1[s[q],q,Z]$ and t is $t_1[r[\ ],q,Z]$ where $r[\ ] = s[q]$.

Remark: A more "tree theoretic" view of transitivity results by imagining the nodes of t_1 colored black (●), those of t_2 not in t_1 colored red (⊗) and those new nodes of t_3 not in t_2 colored yellow (○). Now look at subtrees with red roots (the r_i) and those remaining in t_1 with yellow roots (q_k). It is clear how t_3 can be built from t_1 using r_i and q_k. See figure 3.

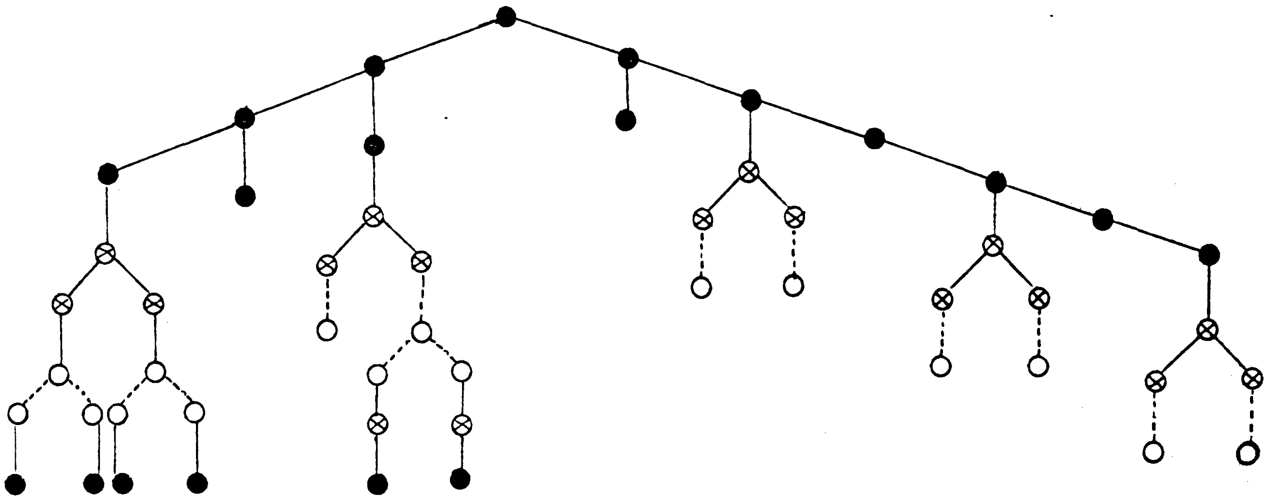


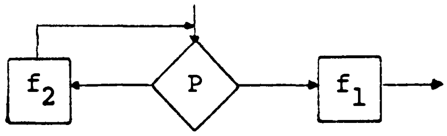
Figure 3.

This proves the claim of transitivity.

(3) We claim $\sqsubseteq$ is anti-symmetric on T_f. That is, if $t_1 \sqsubseteq t_2$ and $t_2 \sqsubseteq t_1$, then $t_1 = t_2$. To see this, suppose that t_1 refines to t_2 via t. Then if $t_1 \neq t_2$, t must be a tree with more than one node (since in the definition of $\sqsubseteq$, $t_i = X_j$ is not allowed). Thus t adds a node to t_1. So t_1 has fewer nodes than t_2. But now it must be possible to go from t_2 to t_1, by substituting a tree $s \neq X_j$. Thus t_2 has fewer nodes than t_1 which is impossible. Q.E.D.

It is not a fact that $\sqsubseteq$ is anti-symmetric on T as the following example shows.

trees for variables. An infinite tree scheme such as Example 1(3),§2, arises from considering the meaning of a flowchart such as



Since we want to allow such flowcharts as abbreviations of algorithms, we also want to allow their substitution for X_i's.

§5. Approximations.

Given an infinite tree scheme t, we can imagine various ways to approximate it by finite trees. Some of these represent the usual ways to specify t as an algorithm by successive refinements. Other ways may not be the usual approximations but may be mathematically useful.

Definition 1: Let T_n be the set of all tree schemes of depth at most n. Clearly $T_n \subset T_{n+1}$.

Definition 2: Given $t \in T$ let $t_n \in T$ be the depth n tree obtained from t by replacing all nodes of depth exactly n by variables (each f_i or p_i gets a distinct variable X_i not used in t) and discarding all nodes of depth greater than n. Call t_n the simple depth n approximation of t.

Example 2:

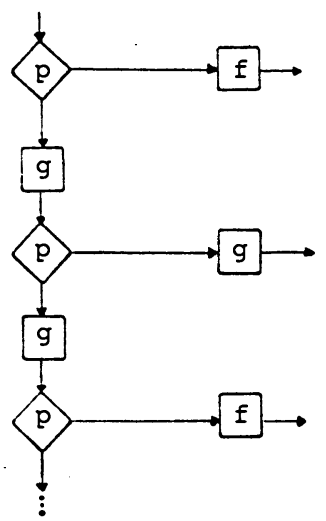
Let t_1 be the tree f g f g f g ... and t_2 the tree g f g f g f .... Consider

s t

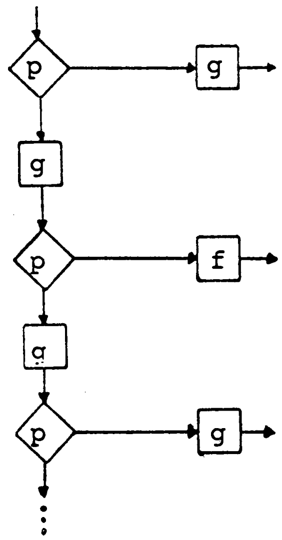
Then $s \sqsubseteq t$ by substituting $\boxed{X} \rightarrow \boxed{f}$ for $\boxed{X}$ and $t \sqsubseteq s$ by substituting $\boxed{X} \rightarrow \boxed{g}$ for X.

The idea clearly applies to trees with exits. We may for example replace

t_1 by



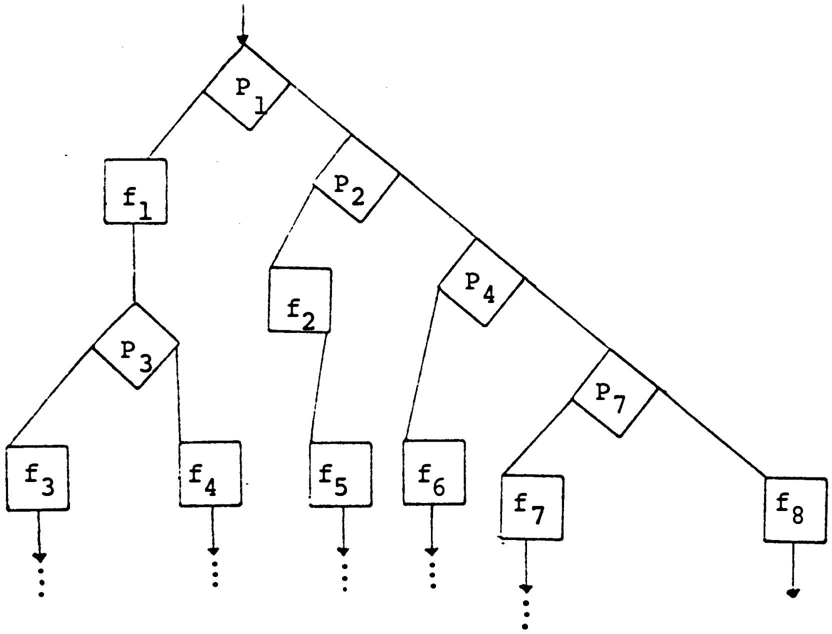
and t_2 by



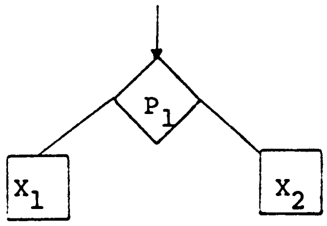
Thus the structure $\langle T, \sqsubseteq \rangle$ is not a partial ordering, only a quasi-ordering or pre-ordering. We call it a QO set, for Quasi-Ordered set.

At this point a word is in order explaining why we insist on allowing infinite tree schemes and the substitution of infinite

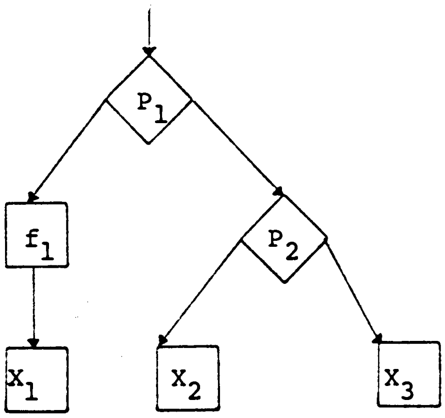
Example 3:



$t_0 := \boxed{X_0}$, $t_1 :=$



$t_2 :=$



<u>Proposition</u> 1: Let t_n be the simple depth n approximation of t. Then $t_0 \sqsubseteq t_1 \sqsubseteq t_2 \sqsubseteq \cdots \sqsubseteq t$.

Proof: Elementary.

Remark: One way to program any algorithm is to specify its simple approximations. This is one of the worst possible ways. It is bottom-up, non-structured and miserable.

When one does not already have an algorithm but is instead trying to specify it, then he may proceed by giving a sequence of ever better approximations. These are captured by the following sequences.

<u>Definition</u> 3: A <u>Scott sequence</u> $d_0, d_1, d_2, \ldots$ is any sequence of finite tree schemes such that

(i) $d_n \in T_n$

(ii) $d_n \sqsubseteq d_{n+1}$

(iii) there is no element of T_n such that $d_n \sqsubset s \sqsubseteq d_{n+1}$, that is, d_n is a <u>maximal approximation</u> to d_{n+1} by trees of depth n .

Example 1:

$$
\begin{aligned}
d_0 &:= X_1 \\
d_1 &:= X_1 \to X_2 \\
d_2 &:= X_1 \to f \to X_2 \\
d_3 &:= X_1 \to f \to X_2 \to g \\
d_4 &:= X_1 \to f \to f \to X_2 \to g \\
d_5 &:= X_1 \to f \to f \to X_2 \to g \to g
\end{aligned}
$$

Example : Consider a Scott sequence for Example 2, §4 .

is extended to $n \rightarrow n'$, then n' exists in s and there is no way to remove it to get back to t .

$d_0 := X_1$	$\bar{d}_0 := X_1$
$d_1 := X_1 \to X_2$	$\bar{d}_1 := X_1 \to X_2$
$d_2 := X_1 \to f \to X_2$	$\bar{d}_2 := X_1 \to g \to X_2$
$d_3 := X_1 \to f \to g \to X_2$	$\bar{d}_3 := X_1 \to f \to g \to X_2$
$\vdots$	

Using simple sequences, we can extend the notion of a Scott sequence and a form of approximation to infinite ~~tree~~ schemes.

Definition 4: Given $s,t \in T$ say that s simply approximates t, $s \sqsubseteq_s t$, iff $s_n \sqsubseteq t_n$ for all n where s_n, t_n are the simple depth n approximations.

The mathematical structure of $\sqsubseteq_s$ on free trees is nicer than that of refinement. We have

Theorem : $\langle T_{free}, \sqsubseteq_s \rangle$ is a partially ordered (p.o.) set.

Proof: We must show that $\sqsubseteq_s$ is an ordering.

(1) $t \sqsubseteq_s t$ is obvious since $t_n = t_n$ for all n, i.e. the nth Scott approximation to t is unique.

(2) $s \sqsubseteq_s t$ and $t \sqsubseteq_s r \Rightarrow s \sqsubseteq_s r$.

We must show $s_n \sqsubseteq t_n$ and $t_n \sqsubseteq r_n \Rightarrow s_n \sqsubseteq r_n$. This is clear since t_n extends the leaves of s_n by a finite amount and r_n extends those of t_n by a finite amount. We substitute the r_n extensions into the t_n extensions to get an extension of s to r .

(3) $t \sqsubseteq_s s$ and $s \sqsubseteq_s t \Rightarrow s = t$.

Claim: $t_n \sqsubseteq s_n$ and $s_n \sqsubseteq t_n \Rightarrow s_n = t_n$.
This is clear since extension occurs at leaves. If leaf node n

§6. Convergence.

We can imagine various attempts to construct algorithms for problems which have no algorithmic solution. Perhaps the problem is only vaguely conceived and is refined and clarified as the program is written. Such attempts are represented by non-converging chains of approximations. For example

<u>Examples</u> 1 : (1) x_1.

$$x_1 \to x_2$$
$$x_1 \to f \to x_2$$
$$x_1 \to f \to x_2 \to g$$
$$x_1 \to f \to f \to x_2 \to g$$
$$x_1 \to f \to f \to x_2 \to g \to g$$
$$\vdots$$

(2)

$$x_1 \to x_2$$
$$x_1 \to x_3 \to x_2$$
$$x_1 \to x_4 \to x_3 \to x_2$$
$$\vdots$$

In example (1) the approximations cannot converge to a tree because the programmer has not provided any exit from his iteration of f. In example (2) he continues to refine the variable x_1. First he decides it must be $x_1 \to x_2$. Then he refines x_1 again to $x_1 \to x_3$, then $x_1 \to x_4$, etc. This does not converge because he cannot decide how to refine that variable.

Our reaction to this situation is to look for conditions which will guarantee convergence; that is, conditions for acceptable programming. Scott in [16] takes another view. To obtain a smoother mathematical theory, he adds limit points for all attempts to program. We consider this approach briefly for comparison.

Definition 1 : A Scott scheme S is any sequence $\{s_i\}$ which is a Scott chain of free finite trees. Scott approximation on such sequences is defined by

$$\{s_i\} \sqsubseteq_S \{t_i\} \text{ iff } s_i \sqsubseteq t_i \text{ for all } i .$$

Let $\mathcal{S}$ be the class of Scott schemata.

We can now prove that $<\mathcal{S}, \sqsubseteq_S>$ is a PO set and is complete in the sense that all ascending chains have limits (but these limits may not be trees). Call such a set a CPO set; it has a unique bottom element denoted $\perp$.

Theorem 1: $<\mathcal{S}, \sqsubseteq_S>$ is a CPO, i.e. every ascending chain has a limit, and $\perp$, the only variable needed, is the minimum element.

Proof: Given a chain $s^0 \sqsubseteq_S s^1 \sqsubseteq_S \ldots$ where each s^j is $<s_i^j>$ we have an array of elements

$$\begin{array}{ll}
s^0 \text{ which is} & s_0^0 \sqsubseteq s_1^0 \sqsubseteq s_2^0 \sqsubseteq s_3^0 \sqsubseteq \;\ldots \\
\sqcap\!| & \sqcap\!| \quad\; \sqcap\!| \quad\; \sqcap\!| \quad\; \sqcap\!| \\
s^1 & s_0^1 \sqsubseteq s_1^1 \sqsubseteq s_2^1 \sqsubseteq s_3^1 \sqsubseteq \;\ldots \\
\sqcap\!| & \sqcap\!| \quad\; \sqcap\!| \quad\; \sqcap\!| \quad\; \sqcap\!| \\
s^2 & s_0^2 \sqsubseteq s_1^2 \sqsubseteq s_2^2 \sqsubseteq s_3^2 \sqsubseteq \;\ldots \\
\sqcap\!| & \sqcap\!| \quad\; \sqcap\!| \quad\; \sqcap\!| \quad\; \sqcap\!| \quad \ddots \\
\vdots & \vdots \quad\;\; \vdots \quad\;\; \vdots \quad\;\; \vdots
\end{array}$$

To define the limit point t , we must specify each component, t_i . Since each tree in column i is of depth i , to this can happen iff $t_j' = t_j$. Q.E.D.

<u>Example</u> 2 : The Scott chain $X_1 \to f^n \to X_2 \to g^n \ldots$ of Example 1 §5 has a limit in $\mathcal{S}$. This limit has the form $f \to f \to \ldots \to g \to g \to \ldots$ which is not a tree since it contains ω-chains at interior points of the structure.

Returning now to the general situation we want to know when an ascending chain has a limit (constructively). Convergence will depend on the overall behavior of the sequence rather than on a term by term criterion.

<u>Definition</u> 2 : A sequence $t_0 \sqsubseteq t_1 \sqsubseteq t_2 \sqsubseteq \ldots$ of trees is called <u>Cauchy</u> iff for each occurrence of a variable X_i we know either that X_i is refined to a tree with a non-variable root or that X_i is never refined beyond a certain term t_k .

Remark. Thus a programming effort is Cauchy if we can control the refinements. If we assign a subtask in the form of a variable X_i to another programmer, we know when he will finish his job.

<u>Theorem</u> 2 : If $t_0 \sqsubseteq t_1 \sqsubseteq t_2 \sqsubseteq \ldots$ is a Cauchy sequence, then we can find an upper bound t for it.

<u>Proof</u>: We show how to construct the upper bound t which we write as $\lim t_i$. We build t level by level starting at the root.

If the root of t_0 is a variable X_i , then determine whether this occurrence of X_i gets refined to a tree with constant root. If it does, say to root f , then take f as a root. Otherwise leave X_i as the root and let t_{k_1} be the define t_i we look in column i. We look to see if any variable appearing in trees in this column can be filled in. More precisely, a non-constructive definition of t_i is given inductively as follows, starting with $m(0) = 0$.

> t_i is obtained from $s_i^{m(i-1)}$ as follows. For each X (i.e. for $\perp$) in $s_i^{m(i-1)}$, if $\exists j$ such that $s_i^j \sqsubseteq s_i^j$ arises by substituting f_k for X, then replace X by f_k to obtain t_i. Let $m(i)$ be the maximum of the j's (maximum row) needed to determine all substitutions.

Note: If a variable X remains in t_i, then X appears in the limit, t.

We must now prove that:

(1) $t \in \mathscr{S}$

(2) $s^i \sqsubseteq_S t$ for all i

(3) if for all $s^i \sqsubseteq_S t'$ then $t = t'$

To show (1) we need only know that t_i is a good i-approximation to t_{i+1}. Clearly $t_i \sqsubseteq t_{i+1}$ by transitivity of $\sqsubseteq$. Moreover, if $t_i \sqsubset q \sqsubseteq t_{i+1}$ and q is of depth i, then for $j \geq m(i)$, by definition of $m(i)$, we know $s_i^j = t_i$. Therefore at $m(i+1)$ we know $s_i^{m(i+1)} = t_i \sqsubset q \sqsubseteq t_{i+1} = s_{i+1}^{m(i+1)}$ which contradicts the assumption that $\{s_k^{m(i+1)}\}_{k=0}^{\infty}$ is a Scott chain.

To prove (2), that $s^i \sqsubseteq_S t$ we must show $s_j^i \sqsubseteq t_j$ for all j. For $j \geq i$ we know this immediately by transitivity of $\sqsubseteq$. For $j < i$ we still know $s_j^i \sqsubseteq t_j$ because if when t_j was being defined, any s_j^k was more refined, then that more refined term was used and $m(j) \geq k$.

To prove (3) we notice that if $s^i \sqsubseteq t' \sqsubseteq t$, then $s_j^i \sqsubseteq t_j'$ for all j and $t_j' \sqsubseteq t_j$ for all j. But clearly for each j

last term in which X_i is refined.

To determine the successor (or successors) of the nodes known at depth n, let t_{k_n} be least term (term of smallest index) in the sequence such that all variables of depth less than or equal to n have either been refined to constant roots or have ceased being refined (we know such a t_{k_n} because $\{t_i\}$ is Cauchy). Notice that the trees t_{k_n} and t are identical up to depth n. This can be proved by induction on depth up to n. Look at the nodes of depth $n+1$ in t_{k_n} and for each occurrence of a variable X_i determine whether it becomes refined to a tree with constant root or ceases to be refined. In the former case, replace X_i by the constant root; in the latter case, leave X_i alone. These are then the nodes of level $n+1$ in the limit tree t. This process defines t. Notice that if s_j are the simple depth j approximations to t, then $t_{k_j} = s_j$.

We now claim that $t_i \sqsubseteq t$ for all i. Given any i there is some j such that $t_i \sqsubseteq t_{k_j}$. We define an appropriate sequence of trees s_m such that t is obtained by extending t_{k_j} by substituting s_m for X_m.

Given variable X_m in t_{k_j}, determine whether it is ever refined. If it is, say to a tree s^1 with root f, then take the root of s^1 as root of s_m. Now proceed to build a tree s_m just as t was built, i.e. by considering the depth 1 variables of s^1, substituting for them to obtain a tree s^2, considering the depth 2 variables of s^2 and substituting, etc. The rule given describes a tree s_m. If s_m is substituted for X_m, then the result produces the subtree of t whose root is the location of variable X_m. This is because s_m and t

are produced in the same way. Q.E.D.

It is easy to see that we cannot in general strengthen this theorem to say that $\lim t_i$ is the <u>least upper bound</u>. Consider the following example.

<u>Example</u> 3: $t := \boxed{X_0} \to f \to g \to f \to g \to \ldots$

$s := \boxed{X_0} \to g \to f \to g \to f \to \ldots$

and let $t_i := \underbrace{\boxed{X_0} \to f \to \ldots \to g \to \boxed{X_i}}_{i}$. Then t_i is a Cauchy sequence converging to t. But also there is an s such that $t_i \sqsubseteq s \sqsubseteq t$.

We can obtain a unique least upper bound if the limit tree is <u>constant</u> i.e. contains no tree variables.

<u>Corollary</u> 1: If $t = \lim t_i$, $\{t_i\}$ is Cauchy and if t is a constant, then t is the least upper bound (lub) of $\{t_i\}$, i.e. if $t_i \sqsubseteq s$ for all i, then $\lim t_i = s$.

In the example above, not only $t_i \sqsubseteq s \sqsubseteq t$ but also $t \sqsubseteq s$. Thus in some sense s and t are "equivalent." This suggests that we see whether this always happens, i.e. does $t_i \sqsubseteq s \sqsubseteq \lim t_i$ imply $\lim t_i \sqsubseteq s$? If so, then we would have a non-unique least upper bound. It does hold as Lemma 1 in §7 shows.

We claim $\phi(x_0) = x_0$. To see this note $\phi(x_0) := \phi(\lim \phi^{(n)}(\bot)) = \lim \phi(\phi^{(n)}(\bot))$ by continuity. Now $\lim \phi(\phi^{(n)}(\bot)) = \lim \phi^{(n+1)}(\bot) = \lim \phi^{(n)}(\bot)$. Hence $\phi(x_0) = \lim \phi^{(n)}(\bot) = x_0$.

Finally we claim that x_0 is the least fixed point. Suppose $\phi(X) = X$. Then since $\bot \sqsubseteq X$ for all X we have by monotonicity

$$\phi(\bot) \sqsubseteq \phi(X) = X \quad \text{so}$$
$$\phi^{(n)}(\bot) \sqsubseteq \phi(X) = X \quad \text{for all } n.$$

Thus $x_0 \sqsubseteq X$ because x_0 is the least upper bound of $\{\phi^{(n)}(\bot)\}$. Q.E.D.

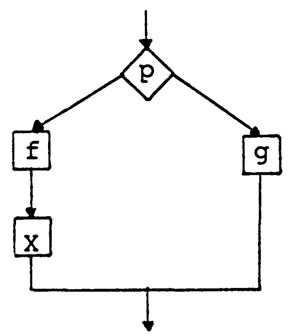
The classical fixed point approach will not apply in T because T is not a CPO. However, a constructive treatment of fixed points can be developed for T which is sufficient for the applications to semantics. The central concept is a Cauchy continuous function.

<u>Definition</u> 2: A function $f: T \to T$ is <u>Cauchy</u> <u>continuous</u> iff

(i) if $\{t_i\}$ is Cauchy, then so is $\{f(t_i)\}$ and $f(\lim t_i) = \lim f(t_i)$.

(ii) for all $t \in T$, $\{f^{(n)}(t)\}$ is Cauchy.

<u>Example</u> 1: Consider the tree



§7. Fixed Points.

In §8 we will apply this work to give a simple definition of the meaning of flowcharts and recursive schemes. Key to such results is an account of fixed points for Cauchy continuous functions. To motivate this approach we review briefly the classical fixed point results as summarized by Scott [16,17,18].

The classical theory of continuous functions in CPO.

<u>Definition</u>^{0} 1: Suppose $\phi: D \to D$ for $\langle D, \sqsubseteq, \perp\rangle$ a CPO, then ϕ is <u>continuous</u> iff $\phi(\lim d_i) = \lim \phi(d_i)$ for ascending chain $\{d_i\}$. $\phi(\ )$ is <u>monotone</u> iff $x \sqsubseteq y$ implies $\phi(x) \sqsubseteq \phi(y)$.†

<u>Proposition</u>^{0} 1: If $\phi(\ )$ is continuous, then it is monotone.

<u>Proof</u>: Given $x \sqsubseteq y$, then $\lim\{x,y\} = y$, so $\phi(y) = \phi(\lim\{x,y\}) = \lim(\phi(x),\phi(y)) = \phi(y)$. Thus $\phi(x) \sqsubseteq \phi(y)$. Q.E.D.

<u>Theorem</u>^{0} 1: If $\langle D, \sqsubseteq, \perp\rangle$ is a CPO and $\phi: D \to D$ is continuous, then ϕ has a least fixed point x_0, i.e.

(i) $\phi(x_0) = x_0$

(ii) if $\phi(x) = x$, then $x_0 \sqsubseteq x$.

<u>Proof</u>: Consider the sequence $\phi(\perp)$, $\phi(\phi(\perp)), \ldots, \phi^{(n)}(\perp), \ldots$. The sequence is ascending because $\perp \sqsubseteq \phi(\perp)$ since $\perp \sqsubseteq x$ for all $x \in D$, thus $\phi(\perp) \sqsubseteq \phi(\phi(\perp)), \ldots, \phi^{(n)}(\perp) \sqsubseteq \phi^{(n+1)}(\perp)$.

Since $\{\phi^{(n)}(\perp)\}$ is ascending and D is a CPO, the sequence has a limit, say x_0. Thus $\phi^{(n)}(\perp) \sqsubseteq x_0$ for all n.

---

† The superscript 0 indicates a classical concept.

This can be regarded as a function $f\colon T \to T$ because whenever X is replaced by a tree $t \in T$, the result is a new tree. We can easily see that this function is Cauchy continuous.

We immediately have

<u>Proposition</u> 2: Every Cauchy continuous function is monotone.

<u>Proof</u>: (just as in the classical case).

We also have a fixed point theorem.

<u>Theorem</u> 2. (Fixed Point): If $f\colon T \to T$ is Cauchy continuous, then we can find t_0 such that $f(t_0) = t_0$, and t_0 is a least fixed point, and is unique if it is a constant.

<u>Proof</u>: Consider $f(X_0)$. By the unique property of variable X_0, namely that $X_0 \sqsubseteq t$ for all trees t, we know that $X_0 \sqsubseteq f(X_0)$. Now also $f(X_0) \sqsubseteq f(f(X_0)) =: f^{(2)}(X_0)$ by monotonicity, and in general $f^{(n)}(X_0) \sqsubseteq f^{(n+1)}(X_0)$. Also $\{f^{(n)}(X_0)\}$ is Cauchy since f is Cauchy continuous.

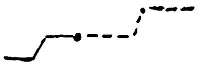
Thus we can find $\lim f^{(n)}(X_0) =: t_0$. Now compute $f(t_0) = f(\lim f^{(n)}(X_0)) = \lim f(f^{(n)}(X_0))$ since f is continuous. But $\lim f^{(n+1)}(X_0) = \lim f^{(n)}(X_0)$, so $f(t_0) = t_0$ as we claimed.

We claim if t_0 is a constant, then it is the least fixed point. We have $X_0 \sqsubseteq t$ for all $t \in T$. Thus $f(X_0) \sqsubseteq f(t)$. So $f^{(n)}(X_0) \sqsubseteq f^{(n)}(t)$ for all n, all X. If $f(t) = t$, then we have $\lim f^{(n)}(X_0) = t_0$ and $f(t) = t$, so $f^{(n)}(X_0) \sqsubseteq t$ for all n.

To conclude $t_0 \sqsubseteq t$ we use the following lemma which is stronger than we need. To conclude uniqueness we use Corollary 1 to

Theorem 2 in §6.

Lemma 1: If t_i is Cauchy and $t_i \sqsubseteq s$ for all i, then $t_i = \lim t_i \sqsubseteq s$ (thus if t is a constant, then $t = s$). (Constructively, if $t_i = \lim t_i$, then $\neg t \not\sqsubseteq s$ and if t is a constant, $\neg t \neq s$.)

Proof: We consider several cases and their subcases.
Case I: Suppose $t_i \sqsubseteq s$ for all $i \geq n$. The suppose further that the substitutions leading to s are never made at an interior variable or a permanent variable, i.e. the substitutions are made for X_i which occurs at a leaf of t_i and is eventually refined to a constant. Then it must be the case that for $t_i = \lim t_i$, $t = s$. Because consider any finite path in s, and consider a substitution in t_n which leads to this path; it has the form  goes to  , but at some t_m, $m>n$, the variable X is refined to a constant, and the constant must be the root of the tree substituted for X because there are no interior variables into which we could substitute to obtain this portion of s. (Any variable which might suffice must occur somewhere along the path and thus must be interior.)
Case II: Again suppose $t_i \sqsubseteq s$ for all $i \geq n$. But now suppose that substitution is made for permanent variables X_j. These X_j are either interior variables or permanent leaves. In any case we distinguish two subcases.

(a) the permanent variable X_j occurs only finitely many times in the limit t.

In this case, let t_{p_j} be the finite tree containing all

Clearly the same substitution must be made for all X_{j_0} since s is a constant (actually the substitution q could be decomposed, say $q = r(p_1,\ldots,p_n)$ and at some occurrence of X_{j_0} one might substitute r, but then the p_i are later substituted, so the composite q could always be used).

Now consider the next earliest variable X_j in the example, Y_1. In general we may have to look within the q substituted for X_{j_0}. Again we can argue that once the X_{j_0} substitution has been made, then the location of the variable X_{j_1} in s is fixed. Therefore whatever substitution is made for X_{j_1} at this earliest occurrence must be made at all occurrences, hence in the limit, otherwise the first occurrence would be damaged.

Thus we see by this argument that in the limit t, we can make the same substitutions as for the approximations t_i.

Q.E.D.

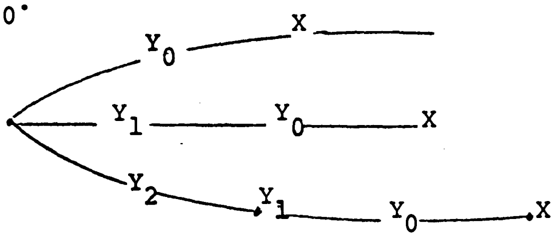
Every tree $t \varepsilon T$ defines a function $t\colon T \to T$ by selecting one tree variable as input. Such functions are sometimes Cauchy continuous. We state this as a theorem. To do so we let $\lambda X_i[t]$ denote the function arising by taking the variable X_i as input. If t has no variables then it is a constant function.

occurrences of X_j. Then in the limit tree t make the substitution q for X_j (where q is such that from $t_{p_j}[X_j]$, $s = t_{p_j}[q]$).

Any path in s which is obtained by substitution into X_j can be obtained from t by the same substitution for X_j.

(b) The permanent variables X_j for which a substitution is made occur infinitely often in the limit t. We assume without loss of generality that there is no variable Y_j which occurs earlier on each path from the root to X_j and for which a substitution to obtain s without using X_j can be made (otherwise we would examine Y_j, not the X_j). In short this means that no variable Y_j can occur earlier than X_j in such a way that the segment from Y_j to X_j is the same on every path.

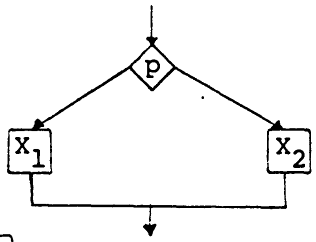
Now select the variable X_{j_0} which occurs earliest, i.e. whose distance to the root is least and for which no variable Y_j used in obtaining s occurs earlier. In the example below, pick Y_0.



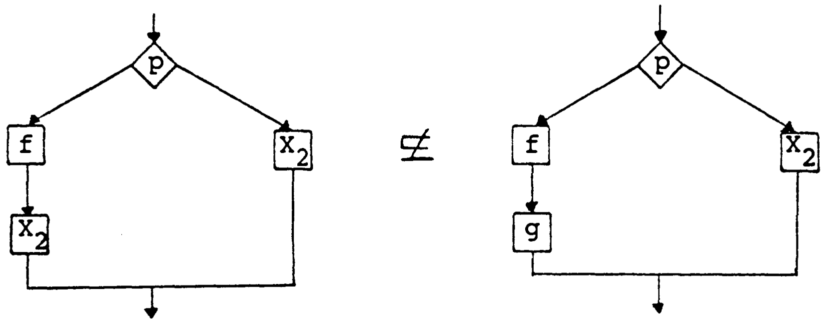
Consider the substitution q made to obtain part of s. Thus in the example, q is the dotted part.

<u>Theorem</u> 3: For all trees $t \in T$, $\lambda X_i[t]$ is Cauchy continuous iff t has at most one variable X_i and a constant root.

<u>Proof:</u> (1) Let $s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \ldots$ be a Cauchy sequence. We must show that $t(s_0)$, $t(s_1)$, $t(s_2)$, ... is a Cauchy sequence if X_i is the only tree variable. If X_i is the only variable, then clearly $t(s_i) \sqsubseteq t(s_{i+1})$ because we can use the refinement giving $s_i \sqsubseteq s_{i+1}$ to show that $t(s_{i+1})$ refines $t(s_i)$. However, if t contains another variable, say X_2, then we can input to t a tree s_1 involving variable X_2 which is refined to give s_2. So while $s_1 \sqsubseteq s_2$, not $t(x_1) \sqsubseteq t(s_2)$ because not all occurrences of X_2 are replaced in $t(s_1)$. For example let t be



and let s_1 be $f \rightarrow \boxed{X_2}$, let s_2 be $f \rightarrow g$. Then $s_1 \sqsubseteq s_2$ but $t(s_1) \not\sqsubseteq t(s_2)$ as the figure shows.



(2) Next we must show that $t(\lim s_i) = \lim t(s_i)$. This can be done by induction on the depth of the trees (since both $t(\lim s_i)$ and $\lim t(s_i)$ are trees) starting at the root and considering the cases that the depth n node is either an $\boxed{X_i}$,

$\boxed{f_i}$ or $\langle P_i \rangle$. We leave the tedious details to the reader.

(3) Finally we must show that $\{t^{(n)}(s)\}$ is Cauchy for any $s \varepsilon T$ as long as t has a constant root. Clearly having a constant root is necessary as the following example shows. Take $t := \to\boxed{X}\to\boxed{f}\to\boxed{X}\to$. The sequence $t(t)$, $t(t(t))$,... is not Cauchy because the first occurrence of X never "settles down."

On the other hand, given that t has a constant root, say $\boxed{f}$ or $\langle P \rangle$, then for any occurrence of X_0 in $t^{(n)}(X_0)$ we know that at $t^{(n+1)}(X_0)$ that occurrence is refined to the root value. Q.E.D.

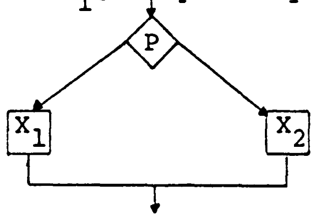
We can talk about a function f: $T \to T$ being Cauchy continuous on a subset D of T . This is the concept we need if we want to regard each $t \varepsilon T$ as a Cauchy continuous function.

<u>Definition</u> 3: f: $T \to T$ is <u>Cauchy continuous on $D \subseteq T$</u> iff

(i) if $\{s_i\}$ is Cauchy and $s_i \varepsilon D$ for all i , then $\{f(s_i)\}$ is Cauchy and $f(\lim s_i) = \lim f(s_i)$

and (ii) $\{f^{(n)}(s)\}$ is Cauchy for all $s \varepsilon D$.

<u>Example</u> 2: The function $\lambda X_1[t]$ given by

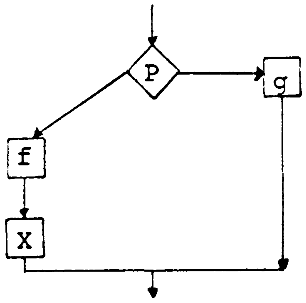


is not Cauchy on T but is Cauchy on the set of all trees not containing the variable X_2 .

<u>Theorem</u> 4: Any tree t with constant root is Cauchy continuous on the set D_t of all trees which do not contain tree variables from

<u>Proof</u> : for reader

But we can also consider a function naturally arising from the flowchart, namely



This function is $t\colon T \to T$. It has a constant root and only one variable, so it is Cauchy continuous. Moreover, $\lim t^{(n)}(X_0)$ is constant, so it is a least upper bound. Thus $t(\ )$ has a least fixed point. The reader can check that this fixed point is indeed the tree obtained by unwinding the flowchart.

The semantics of flowcharts arises as a special case of the semantics for recursion equations because every flowchart can be translated into a set of (right linear) recursion equations which compute the same functional (see [6,7] for a proof).

<u>Definition</u> 1: A set of <u>monadic</u> <u>recursion</u> <u>equations</u> (also known as a <u>de Bakker-Scott</u> <u>scheme</u>, abbreviated <u>dBS</u> <u>scheme</u>) over $\{f_1,\ldots,f_n\}$ consists of a finite set of equations of the form

$$F_i x := \text{IF } P_j(x) \text{ THEN } \alpha x \text{ ELSE } \beta x$$

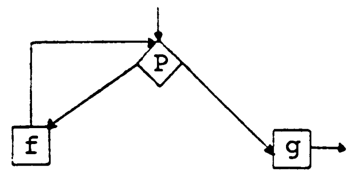
where $\alpha, \beta \in \{f_1,\ldots,f_n,F_1,\ldots,F_p\}^*$. We abbreviate this as $F_i := P_j \to \alpha x, \beta x$. When only the term x is present on the right we also write I for x, as in $P \to I, Ff$, to indicate that the identity functional is being used. The letters F_i are called <u>defined</u> <u>functions</u>, F_1 is the <u>principal</u> <u>function</u>. The F_i are called <u>given</u> <u>functions</u> or base functions.

§8. Semantics of Recursive Schemata.

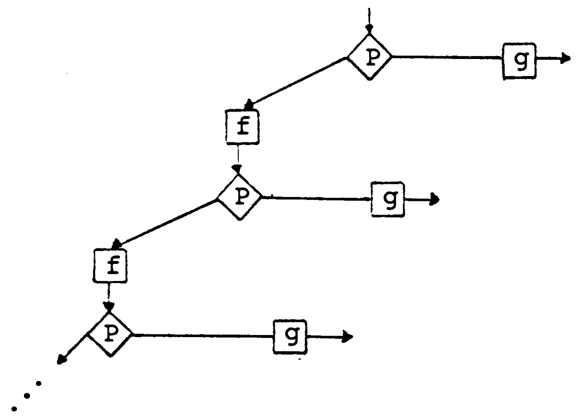
We consider flowcharts and systems of recursion equations as ways to name algorithms. The algorithm itself is a tree scheme which the flowchart or recursion equations denote.

There are numerous ways to show that flowcharts denote trees. For instances, we can "unwind" a flowchart, and it becomes a tree. Likewise given a set E of recursion equations, we can "unwind" them by systematically considering all possible executions. These unwinding methods are inelegant (and especially cumbersome and arbitrary for recursion equations). There is a much better way to assign trees. It is based on the fixed point results of §7. We now examine this fixed point semantics in detail and compare it to other methods. We begin with a very primitive example.

Example 1: Consider the flowchart



This can be "unwound" to form the tree



For a thorough discussion of this fascinating class of schemes the reader should see [7]. We assume familiarity with the standard operational semantics of these schemata.

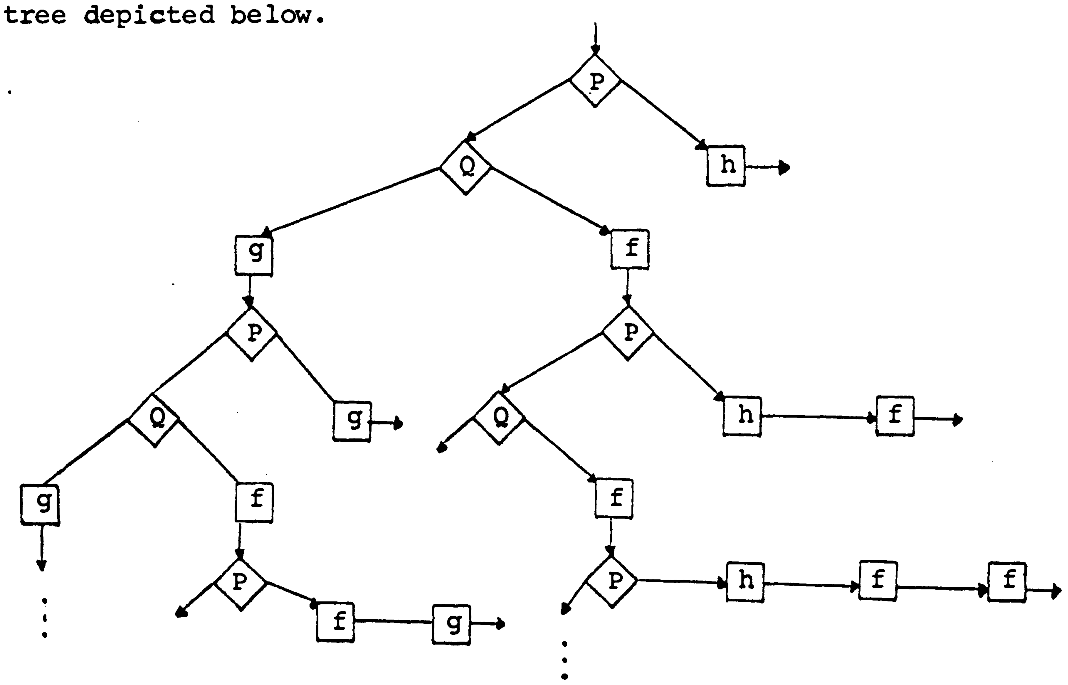
Example 2: $F_1x := P(x) \to hx,\ F_2x$

$F_2x := Q(x) \to fF_1fx,\ gF_1gx$

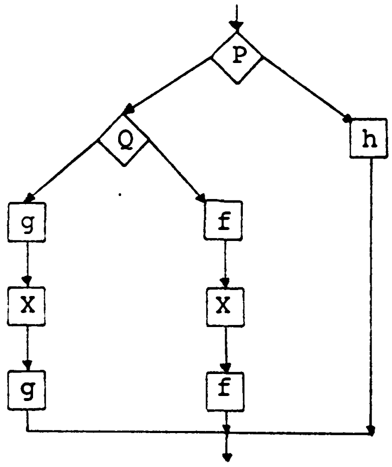
We can write these as a single equation if we nest the conditionals.

$$Fx \ := \ Px \to hx,\ (Qx \to fFfx,\ gFgx).$$

The meaning of this scheme can be found by systematically enumerating paths of execution in the following spirit: "first we must test P, if true then exit with x, otherwise test Q, if true, then compute f and test P, if true then apply f to result and exit, otherwise ...." The result of this rambling is the tree depicted below.



A more elegant way to assign a tree to this recursion scheme is to consider it as a function. Consider the tree



This defines a function $F: T \to T$. The function is Cauchy and has a constant fixed point which is its least fixed point. That fixed point is in fact the correct meaning for the recursion scheme in the sense that the tree and the scheme are strongly equivalent -- they compute the same functional. Below is a diagram of the term $f^{(2)}(X_0)$ in the Cauchy sequence $\{f^{(n)}(X_0)\}$.

Example 3: diagram. (see next page)

The example above suggests the following definition and theorem.

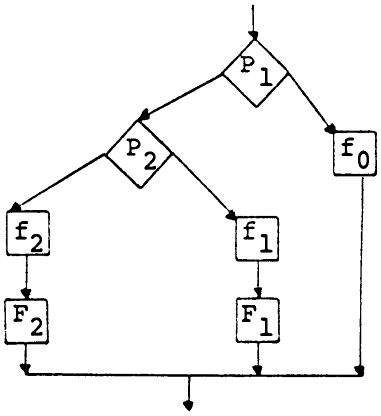
<u>Definition</u> 2: Given a single equation monadic recursion scheme f (with nesting allowed), the <u>derived</u> <u>tree</u> <u>function</u> is the function $f: T \to T$ obtained by replacing the variable F by a tree variable X_0 .

<u>Theorem</u> 1 (semantics): Any single equation recursion scheme is strongly equivalent to the least fixed point of the derived tree function.

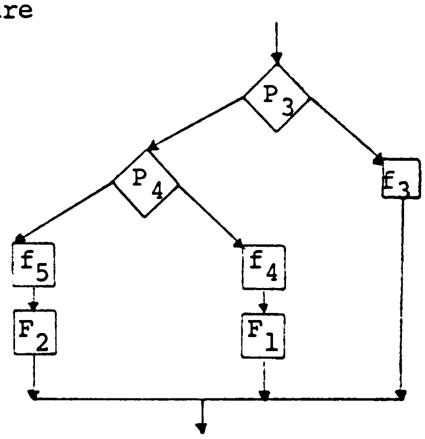
<u>Example</u> 4: Consider the following system of two equations:

$$F_1x \ := \ P_1x \to f_0x, \ (P_2x \to F_1f_1x, F_2f_2x)$$
$$F_2x \ := \ P_3x \to f_3x, \ (P_4x \to F_1f_4x, F_2f_5x)$$

The corresponding tree functions are



F_1 - tree

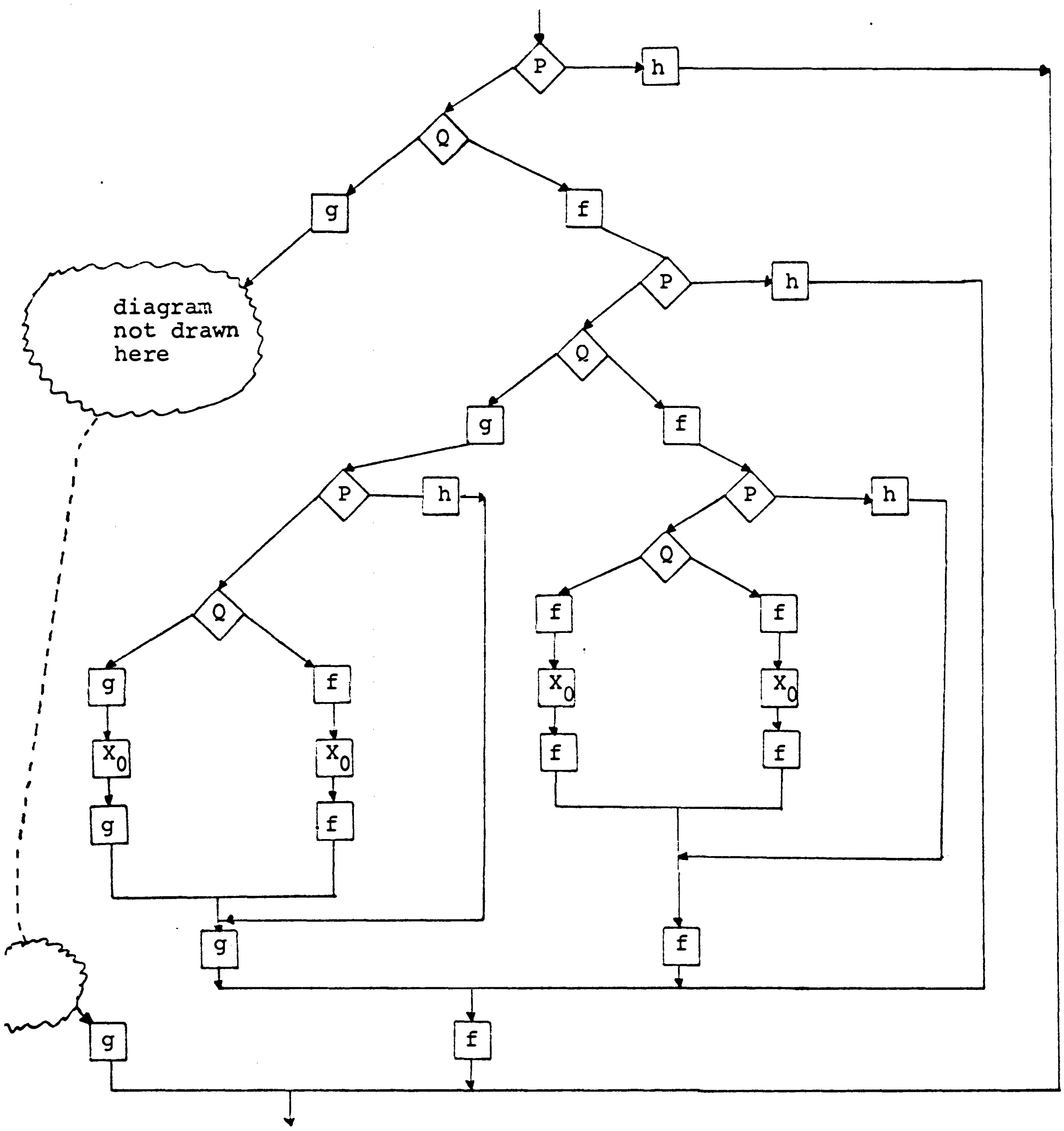


F_2 - tree

We think of F_1, F_2 as variables, X_1, X_2 . These equations are so interconnected that they cannot be replaced by a single equation. In order to "unwind" them we must form two trees which are simultaneously extended. To describe this process mathematically we consider functions from $T \times T$ to $T \times T$. Let the tree functions be $t_1(F_1,F_2)$, $t_2(F_1,F_2)$. We call t_1 the <u>principal tree</u> because it corresponds to the principal function letter. Define now a vector function $t\colon\ T^2 \to T^2$ by $t(F_1,F_2) \ := \ \langle t_1,(F_1,F_2),\ f_2(F_1,F_2)\rangle$. Define iteration in the natural way:

<u>Definition</u> 3: If $F\colon\ T^m \to T^m$ then define

$$f^{(0)}(x_1,\dots,x_n) \ := \ \langle x_1,\dots,x_n\rangle \quad \text{write} \quad f_i(x_1,\dots,x_n) = x$$
$$f^{(n+1)}(x_1,\dots,x_n) \ := \ f(f^{(n)}(x_1,\dots,x_n)_1,\dots,f^{(n)}(x_1,\dots,x_n)_n) \ .$$



<u>Proof</u>: see Theorem 5.

We now want to generalize this approach to any finite system of recursion equations. This will then give us a smooth semantic theory for flowcharts as a special case.

In the above example, $t^{(2)}(F_1,F_3)$ is $<t_1(t_1(x_1,x_2),t_2(x_1,x_2)),\ t_2(t_1(x_1,x_2),t_2(x_1,x_2))>$. We can now examine the sequence $\{t^{(n)}(x_0,x_0)\}$ in T^2 and ask if it has a limit which can be regarded as a fixed point of the vector function $t\colon\ T^2 \to T^2$. Such a fixed point consists of two infinite trees. The tree in the first component, the principal tree, should be the meaning of the recursion equations.

The fixed point approach to the semantics of recursion equations will work if we can extend the results of §6 on convergence and §7 on fixed points to n-tuples of trees, T^n . It is easy to check the following facts.

Definition 4: For $t,s \in T^n$ define $t \sqsubseteq s$ iff $t_i \sqsubseteq s_i$ for all i where $t = <t_1,\ldots,t_n>$, $s = <s_1,\ldots,s_n>$.

Theorem 2: $<T^n, \sqsubseteq>$ is a QO set.

Definition 5: A sequence $t_0 \sqsubseteq t_1 \sqsubseteq \ldots$ in T^n is Cauchy iff the sequence in each component is Cauchy.

Definition 6: A function $t\colon\ T^n \to T^n$ is Cauchy continuous on D if it is Cauchy continuous on D for each component.

Theorem 3 (convergence): Every Cauchy sequence $\{t_i\}$ in T^n converges to $\lim t_i$ which is the least upper bound if each component of the limit is constant.

Proof: just as in Theorem 2, §6.

Theorem 4 (fixed point): If $t\colon\ T^n \to T^n$ is Cauchy continuous, then we can find a $t_0 \in T^n$ such that $t(t_0) = t_0$, and if t_0

is constant in all components then it is the least fixed point.

Definition 7: Given a set of recursion equations E

$$F_i x \;:=\; P_i x \rightarrow \alpha_i x,\ \beta_i x \qquad i=1,\dots,n$$

define the derived tree function as the vector function $E\colon\ T^n \rightarrow T^n$ obtained by taking a derived function for equation F_i in component i (obtained by considering each F_i as a tree variable).

Example 5: In example 4 we defined the derived function

$$t_1(F_1,F_2) \;:=\; P_1 \rightarrow f_0,\ (P_2 \rightarrow F_1 f_1, F_2 f_2)$$
$$t_2(F_1,F_2) \;:=\; P_3 \rightarrow f_3,\ (P_4 \rightarrow F_1 f_4, F_2 f_5)$$

$$t(F_1,F_2) \;:=\; \langle t_1(F_1,F_2),\ t_2(F_1,F_2)\rangle \ .$$

We can now prove

Theorem 5 (semantics): Any recursion scheme E is strongly equivalent to the least fixed point of its derived tree function.

Proof: First we note that the derived tree function is Cauchy continuous because it has a constant root in each component. So the limit of $\{t_E^{(n)}(x_1,\dots,x_m)\}$ exists.

Consider any computation of E , either terminating or non-terminating. This computation is a sequence of function calls and predicate tests such as

$$f_{i_1},\ p_{i_2},\ f_{i_3},\ p_{i_4},\ f_{i_3},\ f_{i_4},\dots$$

We can show that such a sequence is a path in $\lim t_E^{(n)} = t$.

§9. Summary and conclusion.

In this report we have considered three main ideas.

(1) a definition of algorithm as an abstract object distinct from its representation as a program and from its possible mathematical interpretation as a function.

(2) a constructive rendering of a fixed point semantics for recursive procedures.

(3) a definition of refinement, $\sqsubseteq$, as a structural property of programs.

We will provide a brief historical perspective on the new contributions of this paper to each of the topics.

Scott and Strachey in [19] discuss the problem of finding an appropriately abstract notion of algorithm. Work in schema theory such as Engler [6], Luckham, Park and Paterson [11] and others also led in this direction. The new feature of this approach is merely the class of schemes we choose, the tree schemes. It appears that this class has many technical and philosophical virtues, among them are

(i) the trees are the minimal structures needed to describe an algorithm, so they will be part of any adequate definition;

(ii) the scheme level appears to be appropriate for isolating features of the algorithm from its data;

(iii) the class of infinite recursively presented trees is <u>universal</u> (or schematically complete) as discussed in [3, 15, 20].

To prove this proceed by induction on the length of the sequence. We leave the routine details to the reader. Q.E.D.

The class of tree schemes is simpler than Scott's schemes in "The Lattice of Flow Diagrams" [16], yet is richer than the class of flowchart schemes or recursion schemes. Indeed the class of recursively presented tree schemes is universal in the sense of Strong [20], Paterson [15], Constable & Gries [3]. It is, in fact, not difficult to see that the recursively presented tree schemes are equivalent to Strong's deterministic effective functionals and that a natural generalization to non-deterministic trees (allowing non-deterministic branch nodes) gives a class of schemes equivalent to Strong's non-deterministic effective functionals. We will discuss these matters elsewhere.

The constructive fixed point semantics considered in this paper is based directly on Scott's work in "The Lattice of Flow Diagrams" [16], which is in turn based on Scott's 1970 Princeton paper [17]. The origins of this fixed point theory can be found in Kleene's recursion theorem for functionals [9] and other work of logicians. In particular the article by Krieder & Ritchie [10] exploits the fixed point interpretation of recursion equations. However, Scott's 1970 paper was the first work to explicitly use fixed point semantics to solve problems in computer science dealing with the meaning of recursive procedures. His 1970 paper showed the way to a unified treatment of various methods for proving the correctness of recursive procedures.

The version of a fixed point semantics given here differs from Scott's in two minor ways. First, we consider the meaning of a recursive procedure to be an algorithm, not a function in a function space. Second, we present the entire mechanism constructively which helps distinguish the computational aspects of these semantics.

Finally, the notion of refinement used here is new. We are attempting to capture some aspects of the concept of program structure and program development. We shall explore this notion in several directions in forthcoming work. We examine further its role in describing substitutions, its role in unifying various arguments in schematology and its role in describing "structured ways to program."

References

[1] de Bakker, J.W., "Semantics of programming languages," in Advances in Information System Sciences, J.T. Tou (ed.), Plenum Press, New York, 1969.

[2] Cadiou, J.M., "Recursive Definitions of Partial Functions and Their Computations," Ph.D. thesis, Stanford University, Computer Science Department, March 1972, 157 pp.

[3] Constable, Robert L., and David Gries, "On Classes of Program Schemata," SIAM J. Computing, 1, 1, March 1972, 66-118.

[4] Dahl, O.J., Dijkstra, E.W., and C.A.R. Hoare, Structured Programming, Academic Press, London, 1972, 220 pp.

[5] Engler, E. Formal Languages: Automata and Structures. Markham, 1968, 81 pp.

[6] -----, "Structure and Meaning of Elementary Programs," Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics 188, Springer-Verlag, New York, 1971, 89-101.

[7] Garland, Stephen J., and David C. Luckham, "Program Schemes, Recursion Schemes and Formal Languages," JCSS, 7, 2, April 1973, 119-160.

[8] Ianov, Y.I., "The logical schemes of algorithms" in Problems of cybernetics, Vol. I, Pergamon Press, New York, 1960, 82-140.

[9] Kleene, S.C.. Introduction to Metamathematics, D. Van Nostrand, Princeton, 1952, 550 pp.

[10] Kreider, D.L. and R.W. Ritchie, "Predictably Computable Functionals and Definition by Recursion," Ztschr. f. Math. Logik, 5, 1963, 65-80.

[11] Luckham, D.C., D.M.R. Park, M.S. Paterson, "On Formalized Computer Programs," JCSS, 4, June 1970, 220-249.

[12] McCarthy, John. "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems, North-Holland, Amsterdam, 1963 (Braffort & Hirschberg, eds.), 33-70.

[13] Manna, Z. and J. Vuillemin, "Fixed Point Approach to the Theory of Computation," Com. ACM, 15, 7, July 1972, 528-536.

Acknowledgements

I would like to thank the students and visitors in Computer Science 781, Advanced Computing Theory, at Cornell for helping me straighten out these ideas and for making it worthwhile organizing them into notes. My sincere thanks to

Edmund Clarke
Dr. Herbert Egli
Zvi Galil
Kurt Mehlhorn
Steve Muchnick
Michael O'Donnell
Susan Owicki
John Privitera
Morris Siegel
Janos Simon
Marvin Solomon
Professor John Williams

I thank Pauline Cameron for her unfailingly excellent arrangement of these ideas on the typed page.

[14] Park, David, "Fixed Point Induction and Proofs of Program Properties," in Machine Intelligence, 5, American Elsevier, New York, 1969, 59-78.

[15] Paterson, M.S. and C.E. Hewitt, "Comparative Schematology," Conference Record of Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, 1970, 119-128.

[16] Scott, Dana, "The Lattice of Flow Diagrams," in Lecture Notes in Mathematics 188, Symposium on Semantics of Algorithmic Languages, (E. Engeler, ed.), Springer-Verlag, 1970, 311-366.

[17] -----, "Outline of a Mathematical Theory of Computation," Proceedings of the 4th Annual Princeton Conference on Information Sciences & Systems, Princeton, 1970, 169-176.

[18] -----, "Lattice Theory, Data Types and Semantics" in Courant Computer Science Symposium 2: Formal Semantics of Programming Languages, (R. Rustin, ed.) Prentice-Hall, Englewood Cliffs, 1972, 65-107.

[19] Scott, Dana, and Christopher Strachey, "Toward a Mathematical Semantics for Computer Languages," Proceedings of the Symposium on Computers and Automata, Polytech. Institute of Brooklyn, New York, 1971, 19-46.

[20] Strong, H.R., "High Level Languages of Maximum Power," Conference Record 1971 - 12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, October 1971, 1-4.