RELATIONS BETWEEN RECURSIVE DEFINITIONS
AND THEIR MEMORY EFFICIENT IMPLEMENTATIONS.

by: M. C. Paull
January 1977
DCS-TR-#56

Dept. of Computer Science
Rutgers, The State University of N.J.
New Brunswick, New Jersey

This work was supported by Grant No. DAHC15-73-G-6
The Advanced Research Projects Agency.
RELATIONS BETWEEN RECURSIVE DEFINITIONS
AND THEIR MEMORY EFFICIENT IMPLEMENTATIONS

Abstract

Typically there are significant differences between the initial formulation of an algorithm and its ultimate implementation. For example the minimum path between two nodes in a weighted di-graph can be found by enumerating all paths between the two nodes and choosing the smallest. This approach can easily be formulated as a recursively defined function, which may in turn be implemented in a standard way. This is significantly different than Dykstra's algorithm, the favored shortest path implementation. On the one side, close to the problem statement, then there is an initial, simply formulated, but, often inefficient algorithm. On the other side, nearer to the final implementation, is an efficient algorithm. The study of the connection between these two is the subject of this paper.

It will be assumed that the initial formulation of an algorithm is as a recursive definition and that this definition is in a standard form. The recursive definition though sufficient to provide the value of the function anywhere in its domain is non-deterministic as to which of a variety of sequential implementations are to be used to determine that value. The variety of implementations correspond to the various orders of substitution which are equally valid in evaluating such a definition. Some orders of evaluation become possible only if the primitive functions which enter into the recursive definition have appropriate properties. Different orders of evaluation will result in different memory requirements, but will not cause significant time differences in the resultant implementations. This dependence of memory requirements on the order of evaluation is the main subject of this paper.

Note that the material in this paper with the exception of that after page 30, which covers examples and some discussion of a system for automatically producing good algorithms, is identical material appearing in DCS-TR-57.
1. INTRODUCTION

Typically there are significant differences between the initial formulation of an algorithm and its ultimate implementation. For example the minimum path between two nodes in a weighted di-graph can be found by enumerating all paths between the two nodes and choosing the smallest. This approach can easily be formulated as a recursively defined function, which may in turn be implemented in a standard way. This is significantly different than Dykstra's algorithm, the favored shortest path implementation. On the one side, close to the problem statement, then there is an initial, simply formulated, but often inefficient algorithm. On the other side, nearer to the final implementation, is an efficient algorithm. The study of the connection between these two is the subject of this paper.

It will be assumed that the initial formulation of an algorithm is as a recursive definition and that this definition is in a standard form (to be given). The standard form was chosen because, firstly, it is one which, in our experience, has frequently arisen naturally as an initial algorithm formulation. Secondly the chosen form lends itself nicely to an overview of a variety of possible implementations of the algorithm thus formulated. The recursive definition though sufficient to provide the value of the function anywhere in its domain is non-deterministic as to which of a variety of sequential implementations are to be used to determine that value. The variety of implementations correspond to the various orders of substitution which are equally valid in evaluating such a definition. Some orders of evaluation
become possible only if the primitive functions which enter into the recursive definition have appropriate properties. Different orders of evaluation will result in different memory requirements, but will not cause significant time differences in the resultant implementations. This dependence of memory requirements on the order of evaluation is the main subject of this paper.

Related Work

The work reported here is in an area of study in which there have been a number of significant publications. Strong has identified a class of recursive definitions for which memory efficient implementations [5,6] (called 'flowcharts') are available. This class is defined in terms of a recursive scheme whose constituent primitive functions are virtually unrestricted. If the properties of these primitive functions are restricted somewhat, a wider class of recursive definition forms will yield similar memory
efficient implementations. Such restrictions are considered here because they arise naturally in practice. So this aspect of the work can be considered an extension of Strong's results.

Burstable and Darlington studied properties of recursive definitions whose existence allows efficient implementation, with one objective being the incorporation of a search for such properties in an optimizing compiler. Later Burstall and Darlington extended this study to consideration of transformations of recursive definitions which are likely to produce better implementations. The spirit of our work here is largely in tune with that of these investigators with some significant differences in emphasis and in the particular properties studied. Our emphasis has been mainly on understanding the complete set of properties which allow the transformation from an initial recursive definition to the best algorithms actually known and to the proof of this connection. Thus we tend to consider a few relatively complex sets of properties and transformations as opposed to many simple ones. We also study mainly one form of first order\(^1\) recursive definitions, rather than the many forms they consider.

The remainder of this introduction is devoted to a sketch of the definitions and results to be detailed in the body of the paper.

---

\(^1\) First-order means a definition in which the defined function symbol never appears nested on the right.
Appendix I contains a summary of most of the notation used in the paper. (This notation is also defined on first use in the paper.)

The Standard Form

This paper concerns the implementation of recursive definitions of a function $f(X)$ in a class $F$ in which every definition has the following form:

\[
\begin{aligned}
  f(X) &= q(X) \\
  f(X) &= w(f(o_1(X)), \ldots, f(o_m(X))(X)) \text{ if } T(X) \\
  &\text{initially } X \in D_f \quad \text{(domain of function } f) 
\end{aligned}
\]

where the data structure $X \in D_f$, primitive functions $w, q, o_1 \text{ to } o_m$, and predicates $T$ in the definition collectively designated by the tuple $<D, w, q, o_1 \text{ to } o_m, T>$ must be constrained so as to make it a 'substitutionally solvable' definition.

A definition is 'substitutionally solvable' if for each $d \in D_f$ the sequence of expressions resulting from substitution for forms $f(a)$ (where $a$ is any expression) using $l$, which starts with $f(d), l < l \ldots$ and next produces $w(f(o_1(d)), \ldots, f(o_m(d))(d))$, etc. has the properties:

1. It is always possible to evaluate $T(a)$ and if $T(a)$ is false it is always possible to evaluate $m(a)$, and $o_i(a)$ for $1 \leq i \leq m(a)$.
(2) Independent of the order of substitution for the different appearances of the form \( f(a) \), after the same finite number of such substitutions, a 'terminal' expression will be obtained in which, for every appearance \( f(a) \), \( a \) is terminal (i.e. \( T(a) \) is true) and \( q(a) \) can be evaluated.

(3) The function \( w \) is defined so as to make it possible to evaluate the terminal expression in any order consistent with its parentheses structure.

The tuples \( <O,w,q,Q,M,I> \) which satisfy the above constraint are members of the set \( V \). The set of definitions of form \( I \) which satisfy these constraints constitute the recursive scheme \( F(V) \).

This form of definition often arises in practice as an initial solution to an algorithm design problem, particularly when the problem can be viewed as requiring an enumeration or an enumeration followed by a selection (search). The examples of recursive definitions in \( F(V) \) given below arose from adopting such a point of view. Their structure can be easily seen by evaluating them for some small initial values of their arguments.

Examples:

Ex. 1.1 If \( f(N) \) is to be the set of all \( n \) bit binary numbers (let \( N \) be the set of positive integers), then:

\[
X \in \{ \langle \alpha, n \rangle \mid \alpha \text{ a string of 0's and 1's, } n \in N \}
\]

\[
\begin{align*}
  f(X) &= f(\alpha, n) = \{ \alpha \} & \text{if } n = 0 \\
  f(X) &= f(\alpha, n) = f(\alpha \langle 0 \rangle, n-1) \cup f(\alpha \langle 1 \rangle, n-1) & \text{if } n > 0 \\
  \text{where } \langle \rangle \text{ is string concatenation, and } \cup \text{ set union.}
\end{align*}
\]

\( X \) initially \( \{ \langle \lambda, n \rangle \mid n \in N \} \)

Then ex. \( f(\lambda, 2) = f(\langle 0 \rangle, 1) \cup f(\langle 1 \rangle, 1) = (f(\langle 0, 0 \rangle, 0) \cup f(\langle 0, 1 \rangle, 0)) \cup f(\langle 1 \rangle, 1) \)

\( = \{ \langle 00 \rangle \} \cup f(\langle 0, 1 \rangle, 0) \cup f(\langle 1 \rangle, 1) = \text{ etc.} \)
Ex. 1.2 If \( f(X) \) is the set of all permutations of the first \( n \) integers,
\[
X \in \{<n, \alpha> | n \in \mathbb{N}, \alpha \text{ is a string of positive integers}\}
\]
\[
f(X) = \begin{cases} 
\{n\} & \text{if } n = 0 \\
\text{and if } p = |\alpha| = \text{the length of } \alpha, \\
f(X) = f(n, \alpha) = f(n-1, a[0 \hat{+} n]) \cup ... \cup f(n-1, a[p \hat{+} n]) & \text{if } n > 0 \\
\text{where } a[p \hat{+} n] \text{ is an inserting function; i.e.} \\
\text{if } \alpha = <a_1, ..., a_p> \text{ then } a[p \hat{+} n] \text{ is the result of inserting the} \\
\text{integer } n \text{ after component } a_i \text{ in } \alpha \text{ or is } <a_1, ..., a_{i-1}, n, a_i+1, ..., a_p>. \\
\end{cases}
\]
\( X \) is initially \( \in \{<n, \lambda> | n \in \mathbb{N}\} \)

Then \( f(2, \lambda) = f(1, <2>) = f(0, <12>) \cup f(0, <21>) = \{<12>, <21>\} \cup f(0, <21>) \)

Ex. 1.3 \( f(X) \) is the string of moves (each a pair of numbers \(<a, b>\)
meaning move a disc from pin \( a \) to pin \( b \)) necessary to optimally
solve the, now classical, Tower of Hanoi puzzle. To move \( n \) discs initially on pin 1 to pin 2:
\[
X \in \{<x, y, z, n> | <x, y, z> \text{ is a permutation of } <1, 2, 3>, n \in \mathbb{N}\}
\]
\[
f(<x, y, z, n>) = <x, y> & \text{ if } n = 1 \\
f(<x, y, z, n>) = f(<x, z, y, n-1>) \cup f(<x, y, z, 1>) \cup \\
f(<z, y, x, n-1>) & \text{ if } n > 1 \\
X \text{ is initially } \in \{<1, 2, 3, n> | n \in \mathbb{N}\}
\]

Algorithms to Implement Definitions in \( F(V) \) which are Efficient in Use of Memory

An 'algorithm scheme' define a set of algorithms in a manner
analogous to that used in defining a recursive scheme like \( F(V) \). In this paper
algorithm schemes generally will involve standard assignment and conditional
statements using the same unspecified set of data-structures \( D \), primitive
functions \( w, q, o, m \) and predicate \( T \)(designated by the tuple \(<D, w, q, o, m, T>\))
used in defining the recursive scheme \( F(V) \). If we constrain the selection of tuples to be a member of a set \( V \), the set of algorithms thus defined is designated \( S(V) \) and a particular algorithm \( \in S(V) \), corresponding to a tuple \( v \in V \) is designated \( S(v) \). The recursive and algorithm scheme \( F(V) \) and \( S(V) \) are equivalent iff for each \( v \in V \), \( F(v) \) is equivalent to \( S(v) \). A recursive function definition \( F(v) \) and an algorithm \( S(v) \) are equivalent if, with domain \( D \) in \( v \), for every \( d \in D \), the value of \( f(d) \) as computed with recursive definition \( F(v) \) = value of the result of running the algorithm \( S(v) \) with \( d \in D \) as its initial value.

The main purpose of this paper is to show that for a set \( V' \), built from \( V \) by constraining the function \( w \) to be 'associative' and the set of functions \( O \) to have an 'inverse', there is an algorithm scheme \( S(V') \) equivalent to \( F(V') \) which is particularly efficient in its use of memory. The algorithm scheme available when these conditions are satisfied is given in figure 2.2. The algorithm scheme \( S(V') \) is given in terms of the data-structures, primitive functions and transformations of these primitive functions (inverse of \( O \) for example) which are immediately available under the assumption of the existence of an 'inverse', that appear in the equivalent recursive scheme \( F(V') \).

For many of the recursive definitions in the class \( F(V') \), the equivalent member of the class \( S(V') \) - which can be obtained mechanically from the recursive definition is the 'good' algorithm usually used to realize that definition. Thus corresponding to example 1.1, the algorithm obtained by instantiation of that particular \( D, w, q, \alpha, m \) and \( T \) in \( S(V') \) is one in which:

---

2 These terms are defined in section 2. An inverse operation plays a similar role in [6]. Our 'inverse', however, is different, having been independently developed [7,8] in combination with associativity to delineate another class of definition with efficient implementations. The result is in Theorem 2.2.
First a string of n 0's is formed and outputted - being the first binary number produced, then, because the rightmost symbol in the string is a 0 it is changed to a 1 and the result outputted. In general, the algorithm remembers the last binary number formed and outputted, say X. The next binary number is formed by a scan of the bits of X starting with the rightmost bit, and changing them by the following scheme. Let b be the bit under scrutiny - if b is a 0 it is changed to a 1 and the result is the next binary number to be outputted - if it is a 1 it is changed to a 0, b becomes the bit in X one position to the right of the current b and the scrutiny is repeated. When the leftmost bit of a number X becomes b and that bit = 1 then the process terminated. In summary this algorithm for producing all n-bit binary numbers, consists simply in 'adding 1' to produce successive members of the set. It is the 'good' algorithm for producing the set. It keeps in memory only the last number produced thus using an amount of storage roughly equal to that required to hold the argument of f in its recursive definition. This is characteristic of all the algorithms in S(V') in relation to the equivalent member of F(V') and is the 'memory efficiency' mentioned.

In a similar way, the algorithm for example 1.2 obtained by instantiation of the primitives that appear in the recursive definition in figure 2.1 produces one permutation at a time. A permutation is produced from the previous permutation by interchange of adjacent terms. This again is the 'good' algorithm for generating permutations.

Creating an Inverse

In examples 1.1 and 1.2, the given 0-functions had an inverse - in example 1.3 the 0-function as given does not have an inverse and thus the
algorithm scheme \( S(V') \) is not available. However, as will be shown\(^3\) - when in a recursive definition, \( f \in F(V) \), the 0-function does not have an inverse - a simple transformation of \( f \) to an equivalent definition, \( f' \), involving an 0-function having an inverse can always be found in \( F(V') \). Thus \( f' \) will have an equivalent in \( S(V') \). This new definition \( f' \) is equivalent to \( f \) in the sense that to each argument \( d \) of \( f \) there is a 'simply' computed argument \( d' \) of \( f' \) such that \( f'(d') \neq f(d) \). Using this transformation, an equivalent definition to that of example 1.3 will be given subsequently, whose equivalent algorithm in \( S(V') \) will produce the moves necessary to solve the Tower of Hanoi problem - one at a time, the only temporary memory necessary being that for a record of the previous move and its number.

Interpretation of Memory 'Efficiency'

Although the 'memory efficient' algorithms of \( S(V') \) are honestly so for the most part, the nature of the memory efficiency can be misleading. The implementing algorithm available when \( w \) is 'associative' and the 0-function has an 'inverse' is efficient in the sense that the memory required is usually of the order of the largest storage required for the argument (also called a data structure) of \( f \) which arises if \( f \) is evaluated by successive substitutions.

Usually this largest data-structure for which memory need be provided requires a small amount of memory relative to the total of all data-structures produced during the implementation of the definition for a given

\(^3\) Theorem 2.1.
initial data-structure - ex. of the order of a single member of a set
when a set is being enumerated. Even when the 'inverse' does not exist
it can be incorporated as previously noted, leaving the 'memory
efficiency' notion still viable. However there is another way of obtaining
a 'memory efficient' equivalent algorithm which is deceiving and thus worth noting.

This technique involves obtaining a technically correct equivalent
recursive definition of \( f \), say \( f' \) having only one occurrence of \( f' \) on the
right, but in compensation involving much larger data structures \( X' \) and
complex function \( a' \) than the corresponding \( X \) and \( a \) of \( f \). That is, for each
definition of form II there is an equivalent definition
of the form:\(^4\)

\[
\begin{align*}
\text{II} & : \\
 f'(X') &= a'(X') \quad \text{if } T(X') \\
 f'(X') &= w(f'(a'(X'))) \quad \text{if } \neg T(X') \\
\text{Initially, } X' & \in D_f \\
\end{align*}
\]

By equivalent, we mean that there is a 1-1 correspondence
\( g \) between \( D_f \) and \( D_{f'} \) so that for each \( d \in D_f \):

\[
f(d) = f'(g(d))
\]

If \( f' \) has an inverse then it can be realized in the same memory efficient
manner as other definitions in \( F(V') \) and if not it can easily be modified
so as to have one while still keeping the result in the form
of II. Memory efficiency, however, means that the memory requirement
will not exceed the size of the largest data-structure which arises as an argu-
ment of \( f \) during evaluation of \( f' \). But in this equivalent definition that

---

4 The two classical ways this can be done are by constructing a general
breadth-first or depth-first algorithm to implement the recursive definition
of form I and then equivalently giving these as recursive definitions.
data-structure is typically much larger (often exponentially) than that which could arise in the original definition.

The term 'memory efficiency' as used here then requires caution in its application.

The paper concludes with a number of examples of the "good" algorithms which can be obtained by use of the "inverse" flowchart 1, and a brief discussion on mechanically determining whether a given recursive definition has an inverse.
2. MEMORY EFFICIENT IMPLEMENTATIONS

Definition of Standard Recursive Scheme \( F \):

Consider the set \( F' \) of all functions \( f \) that can be defined as follows:

Def. 2.1

\[
\begin{align*}
\text{form} & \quad f(X) = q(X) \quad \text{if } T(X) \\
& \quad f(X) = \omega(f(o_1(X)), \ldots, f(o_m(X))(X)) \quad \text{if } \neg T(X)
\end{align*}
\]

where the primitive functions and predicates which are used in the definition are weakly constrained as to the nature and extent of their domains and ranges. \( D_f \) is the set of initial data-structures and may be any set. Other sets must be included in some of the domains of some of the primitive functions. These other sets are defined recursively, using the primitive functions. First these sets are named and their relation to the primitive functions given, then they are defined.

\( m \) is a function whose domain must include the set \( \Delta_f \) and whose range is the positive integers \( \geq 1 \). \( m(X) \geq 1 \) for all \( X \in \Delta_f \).

\( \omega \) is a set of functions \( \{o_1, o_2, \ldots\} \).

The domain of \( o_1 \) must include the set \( \delta^i \). \( \Delta_f \) is the union of all the functions in \( o_1 \) and is called the domain of the \( \omega \)-function.

The range of \( o_1 \) must include the set \( p^i \).

The union of the sets \( p^i \) of all the functions in \( \omega \) is the range of the \( \omega \)-function and is called \( P_f \).

\( T \) is a predicate whose domain includes \( D_f \cup P_f \). Its range is \( \{\text{true, false}\} \).

\( q \) is a function whose domain must include \( Q_f \). Its range may be any set, say \( W_f \).
\( w \) is a function whose range is called \( W_f \) and whose domain must include
\[
W_f \cup w_f.
\]
The sets named above are defined as follows (the subscript \( f \) is
dropped where it is not essential):
\[
\Delta^1 = \{ d \mid d \in D \text{ and } \overline{T}(d) \}; \text{ and for } j > 1
\]
\[
\Delta^j = \{ o_{\lambda}(X) \mid X \in \Delta^{j-1} \text{ and } i \leq m(X) \text{ and } \overline{T}(o_{\lambda}(X)) \}
\]
\[
\Delta = \bigcup_{i=1}^{\infty} \Delta^i
\]
The set \( \Delta^i \) of \( o_{\lambda} \in \Delta \) is:
\[
\delta^i = \{ X \mid X \in \Delta \text{ and } i \leq m(X) \}
\]
The range of \( Q \) is:
\[
P = \{ o_{\lambda}(X) \mid X \in \Delta \text{ and } i \leq m(X) \}
\]
The range \( \delta^i \) of \( o_{\lambda} \in \Delta \) is:
\[
p^i = \{ o_{\lambda}(X) \mid X \in \delta^i \}
\]
The set of terminal data-structures \( Q \) is:
\[
Q = P - \Delta
\]
The set \( W \) is defined as follows:
\[
W^1 = \{ w(X_1, \ldots, X_n) \mid X_i \in W_i, n = \text{a positive integer} \}; \text{ for } j > 1,
\]
\[
W^j = \{ w(X_1, \ldots, X_n) \mid X_k \in W^k, k(j); n \in N \}
\]
\[
W = \bigcup_{i=1}^{\infty} W^i
\]

If in addition to being a member of the set \( F' \), a recursive
definition is substitutionally solvable as defined below it is a member
of the set \( F \). We need some preliminary definitions.

If \( <i_1, \ldots, i_n> = 1 \) is a sequence of integers then \( o_{<i_1, \ldots, i_n>}(X) = \)
o_{\lambda}(X) is an abbreviation for \( o_{\lambda_1, o_{\lambda_2}(o_{\lambda_3}(X)) \ldots}; o_{\lambda}(X) = X. \)
A length 1 sequence of integers $<i_1>$ is applicable to a data-structure $X \in \Delta_f$ if $i_1 \leq m(X)$. A length $n$ sequence of integers $<i_1, \ldots, i_n>$ is applicable to a data-structure $X$ if $<i_1, \ldots, i_{n-1}>$ is applicable to $X$ and $<i_n>$ is applicable to $o_{<i_1,\ldots,i_{n-1}>}(X)$.

$I(X)$ is the set of all integer sequences applicable to $X \in \Delta_f$.

$f$ is substitutionally solvable iff $\forall d \in D$, $I(d)$ is finite.

Note that if $I(X)$ is finite it cannot contain an infinite sequence, because it always contains all prefixes of any sequence it contains.

This completes our definition of $F$.\footnote{As defined here is the same as $F(V)$ as defined in the introduction.} Next we give some simple consequences of the definition which will be used later. First, the substitutionally solvable property that $d \in D$, $I(d)$ is finite can be extended to any $X \in \Delta_f$. This is done in lemmas 2.1 and 2.2.

### Simple Properties of $F$\footnote{As defined here is the same as $F(V)$ as defined in the introduction.}

**Lemma 2.1:** If $f \in F$ and $X \in \Delta_f$ then an integer sequence $I \in I(d)$ and a data-structure $d \in D$ such that $o_f(d) = X$.

**Proof:** If $X \in \Delta_f$ then obviously there exists some $c$ (at least 1) such that $X \in \Delta^c$. The lemma is proven by induction on the sets $\Delta^j$. Assuming there is a length $k$-1 sequence $I_Y$ for each data-structure $Y \in \Delta^{k-1}$ and $d \in D$ such that $I_Y = o_f(d) = Y$. Then it follows, by definition of $\Delta^k$ that if $X \in \Delta^k$ then $X = o_f(Y)$ for some $i \leq m(Y)$, and $Y \in \Delta^{k-1}$. Thus $X = o_f(o_{I_Y}(d)) = o_{<i>_{I_Y}}(d)$. Since also $D = \Delta^1$, and $o_f(d) = d$ for each $d \in D$, the proof is complete.

**Lemma 2.2:** If $f \in F$ and $X \in \Delta_{\Delta_f}$ then $I(X)$ is finite.

**Proof:** From the previous lemma the data-structure $X = o_{<i>}(d)$ for some $d \in D$ and integer sequence $I$. Therefore $I(d) \geq$ the
set consisting of 1 concatenated with each member of \( I(X) \).
Thus if \( I(X) \) is not finite, \( I(d) \) cannot be finite but this
contradicts the condition that \( f \in F \) is substitutionally
solvable.

Another consequence of the definition of \( F \) is that the data-
structures in \( \Delta_f \) can be usefully ordered in another, almost reverse,
manner than the ordering by membership in the subsets \( \Delta^j \). In most of
the subsequent inductive proofs, induction will be carried out on this
ordering.

Ordering the Data-Structures in \( \Delta \) (Remoteness):
For any function \( f \) in \( F \):

We say a data-structure \( X \) in \( \Delta_f \cup Q_f \) is of remoteness 0 (or is
terminal) if \( X \in Q_f \).

We say a data-structure \( X \) in \( \Delta_f \cup Q_f \) is of remoteness \( n \) if:

(1) \( \exists i: i \leq m(X) \) and \( \sigma_i (X) \) is of remoteness \( n-1 \) and

(2) \( \forall i: i \leq m(X) \) implies \( \sigma_i (X) \) is of remoteness \( n-k \) and \( k \geq 1 \).

Lemma 2.3: If \( f \in F \), then there is a function \( r \) with domain \( \Delta_f \cup Q_f \)
such that if \( X \in \Delta_f \cup Q_f \), then \( r(X) \) is the remoteness of \( X \).

Proof: For each \( X \in \Delta_f \cup Q_f \) let \( r(X) \) be the maximum of the length
of all the sequences in \( I(X) \). For each \( X \in \Delta_f \cup Q_f \), \( X \) is of

---

6 Alternately this can be phrased 'of remoteness < \( n \)'. 
remoteness \( r(X) \). This is shown by induction. If \( T(X) \) then
\( I(X) \) is empty and \( r(X) = 0 \). Assume that if \( r(X) < n \), \( X \) is
of remoteness \( r(X) \). Let \( r(X) = n \), i.e. there is a longest
sequence of length \( n \), say \( I = \langle i_1, \ldots, i_n \rangle \) in \( T(X) \). Let
\( o_{i_1}(X) = Y \). Then \( I' = \langle i_2', \ldots, i_n \rangle \) is in \( T(Y) \). Furthermore,
no sequence applicable to \( Y \) is longer than \( I' \) because other-
wise \( I \) could not have been a longest sequence in \( T(X) \). So
\( r(Y) = n-1 \) and \( Y \) is of remoteness \( r(Y) = n-1 \). Therefore,
since \( o_{i_1}(X) = Y \) and for all \( j \neq i_1 \), \( j \in m(X) \), \( r(o_j(X)) \leq n-1 \),
\( X \) is of remoteness \( r(X) = n \) by definition of remoteness.

---

Properties of \( f \in F \) Sufficient for Memory Efficient Implementations

An efficient implementation becomes
available when the recursive definition \( f \in F \) has some special properties.
These properties are now defined.

**Associativity:** Associativity has the usual meaning here. The function
\( w \) is associative if:

\[
w(a_1, a_2, \ldots, a_m) = w(w(a_1, a_2), a_3, \ldots, a_m) \text{ for } m \geq 3
\]

\( w = \text{minimum}, \text{sum}, \text{catenation and union} \) provide examples of \( w \)-functions
with this property. In each case one can compute \( w(a_1, \ldots, a_m) \) as follows:

\[
x + k
\]

For \( i = 1 \) to \( m \)

\[
y + w(x,a_i)
\]

\[
x + y
\]

End
thus requiring at any one time memory for at most 2 copies of the result of \( w(a_1, \ldots, a_j), j \leq m \). If \( w \) is the function minimum, this memory does not increase on the number, but only on the value of its arguments, \( a_i \). If \( w \) is catenation, sum, or union the memory required will increase, albeit at different rates, with the number of arguments. There is, however, a significant difference in use of the memory, between a computation of catenation and of union. To obtain catenate \((a, b)\), \( b \) needs only be attached at the end of \( a \). To obtain the union \((a, b)\), a must be searched for an occurrence of a member of \( b \). If \( a \) represents the result of a previous computation then in the union case it is necessary to re-access this memory whereas this is not necessary in the catenation case. This is an important consideration because memory that is not re-accessed can be located in areas of memory (disc) which need not be easy to access (as is core). The temporary memory requirements for the implementation of a function then do not depend on the usual mathematical properties of that function only, but also depend on the means available for accessing the memory. Nevertheless, for compactness our results are given in terms of the usual mathematical properties—so caution is needed in their interpretation.

**Uniform Inverse:**

Consider a set of functions \( H = \{ h_1, \ldots, h_M \} \). Let \( D_i \) be the domain over which \( h_i \) is defined and let \( R_i \) be the corresponding range of \( h_i \). Then we will say \( D = \bigcup_{i=1}^{M} D_i \) is the domain of \( H \) and \( R = \bigcup_{i=1}^{M} R_i \) is its range.

The set of functions \( H \) is said to have a uniform inverse on the domain

---

7 It is also true that there may be some advantage in time efficiency in one grouping of the arguments of \( w \) over another though both give the same result when \( w \) is associative. An example of such a function is merge, i.e. merge\((a_1, \ldots, a_m)\) in which \( a_j \) are each finite sorted sets of numbers.
If the functions $H^{-1}$ and $i_H$ defined as follows both exist on the set $\mathcal{R}$.

If $r$ is any member of $\mathcal{R}$:

$$H^{-1}(r) = d$$

where $d$ is the unique $d \in \mathcal{D}$

$$h_i(d) = r$$

for all $i \leq M$ for which $h_i(d) = r$

$$i_H(r) = i$$

where $i$ is the unique index $\exists$

$$h_i(d) = r$$

for all $d \in \mathcal{D}$ such that $h_i(d) = r$

The existence of a uniform inverse of a set of functions depends on properties of the constituent functions $h_i$, their domains and ranges.

The dependences below are a direct consequence of the definitions.

Lemma 2.4: (a) $H^{-1}$ exists iff $\forall i \leq M$: $h_i$ has an inverse, and with $d_1$ and $d_2$ in $\mathcal{D}$, $h_i(d_1) = h_i(d_2)$ implies $d_1 = d_2$.

(b) $i_H$ exists iff $\forall k \geq 1$: $R_i \cap R_{i+k} = \emptyset$ for $k > 0$.

(c) $H^{-1}$ exists if $i_H$ exists and $\forall i \leq M$: $h_i$ has an inverse.

It is possible that either of the functions $H^{-1}$ or $i_H$ exists while the other does not.

The uniform inverse condition though very strong often arises in practice.

If the set of functions $o_1 \in \mathcal{O}$ which appear in a definition $f \in \mathcal{F}$ does not have a uniform inverse then an equivalent definition can always be found which does. This is shown after a short digression required to develop the definition of equivalence.
Equivalence of Recursive Definitions:

Consider two definitions in \( F \):

1. \( f \) on domain \( D \)
   \[
   \begin{align*}
   f(X) &= q(X) & \text{if } T(X) \\
   f(X) &= w(f(o_1(X), \ldots, f(o_m(X)(X)))) & \text{if } T'(X) \\
   \text{initially } X = d & \in D
   \end{align*}
   \]

2. \( g \) on domain \( D' \)
   \[
   \begin{align*}
   g(X') &= q'(X') & \text{if } T'(X') \\
   g(X') &= w'(g(o_1'(X'), \ldots, g(o_m'(X')(X')))) & \text{if } T'(X') \\
   \text{initially } X' = d' & \in D'
   \end{align*}
   \]

If there is a 1-1 correspondence between \( D \) and \( D' \) such that whenever \( d \in D \) and \( d' \in D' \) are two corresponding data-structures \( f(d) = g(d') \) then the two definitions are equivalent. The above correspondence may be extended to one between \( \Delta_f \) and \( \Delta_g \) with \( \delta \in \Delta_f \) corresponding to \( \delta' \in \Delta_g \) by having \( o_i(\delta) \) correspond to \( o_i'(\delta') \) whenever \( \delta \) corresponds to \( \delta' \) and \( o_i(\delta) \) and \( o_i'(\delta') \) are both defined. This is called a structural correspondence. If in addition to such a structural correspondence of \( \Delta_f \) to \( \Delta_g \) the following conditions hold

1. \( T(\delta) = T'(\delta') \)
2. \( q(\delta) = q'(\delta') \) if \( T(\delta) \) (and \( T'(\delta') \))
3. \( m(\delta) = m'(\delta') \) if \( T(\delta) \) (and \( T'(\delta') \))
4. \( w = w' \)

then \( f \) and \( g \) are strongly equivalent. Note that the structural correspondence of \( \Delta_f \) to \( \Delta_g \) need not be 1-1. It will not even necessarily be defined on all members of \( \Delta_f \) and \( \Delta_g \) unless the conditions (1) through (4) are satisfied.
Strong equivalence of two definitions implies that they not only give the same results but also require the same number of substitutions in their evaluation for corresponding initial arguments.

As an example of a strong equivalence, consider the two functions \( f \) and \( g \) each in \( F \):

1. \[
\begin{align*}
& \begin{cases} 
  f(X) = q(X) & \text{if } T(X) \\
  f(X) = w(f(o_1(X)), \ldots, f(o_m(X))) & \text{if } \overline{T}(X)
\end{cases} \\
& \text{initially } X = d \in D
\end{align*}
\]

2. \[
\begin{align*}
& \begin{cases} 
  a) \ g(X,Y) = q(X) & \text{if } T(X) \\
  b) \ g(X,Y) = w(g(o_1(X), h_1(Y)), \ldots, g(o_m(X), h_m(Y))) & \text{if } \overline{T}(X)
\end{cases} \\
& \text{initially } <X,Y> = <d,y_0> \in D' \text{ with } d \in D \text{ and } y_0 \text{ a constant}
\end{align*}
\]

\( H = \{h_1, \ldots, h_M\} \) is a set of primitive functions.

Let data-structure \( d \in D \) correspond to \( <d,y_0> \in D' \). Extend this correspondence to one between \( \Delta_f \) and \( \Delta_g \) by letting \( o_i(X) \in \Delta_f \) correspond to \( <o_i(X), h_i(Y)> \in \Delta_g \) whenever \( X \in \Delta_f \) corresponds to \( <X,Y> \in \Delta_g \) and \( T(X) \) and \( i \leq m(X) \). For example if \( d \in D \) and \( o_i(d) \) is defined then it corresponds to \( <o_i(d), h_i(y_0)> \in \Delta_g \).

For each member of \( \Delta_f \) this correspondence defines a corresponding member of \( \Delta_g \). This follows because every member in \( \Delta_f \) is either in \( D \), for which the correspondence is given explicitly, or it is \( o_i(X) \) for \( X \in \Delta_f \) and \( o_i \) is defined and \( \overline{T}(X) \), in which case the correspondence to a member of \( \Delta_g \) is given since \( o_i(X,Y)'s \) existence just depends on \( X \), because \( m'(X,Y) = m(X), T'(X) = T(X) \).

Conditions (1) through (4) are obviously satisfied for this correspondence in the above definitions. Furthermore, the function \( g(X,Y) \) is
independent of $Y$, its second argument. This is shown inductively as follows. Directly from the definition (2a) we see that $g(X,Y)$ is independent of $Y$ when $(X,Y)$ is of remoteness 0. Its being of remoteness 0 is also independent of $Y$. Referring to (2b), if it is assumed that each term $g(o_1(X),h_1(Y))$ appearing on the right is independent of its second argument then it follows certainly that $g(X,Y)$ on the left of (2b) is independent of $Y$. If the argument on the left side of (2b) is of remoteness $n$ from terminal then all the arguments of terms on the right are of remoteness $< n$ from terminal. Thus the inductive argument is completed concluding that $g(X,Y)$ is independent of $Y$ if $X$ and thus if $(X,Y)$ is of remoteness 0, 1, 2, ..., $n$.

Thus definition (2) can be rewritten removing $Y$ which with $f$ replacing $g$ is the same as (1). Therefore:

Lemma 2.5 $g$ and $f$ above are strongly equivalent.

Since the value of $g(X,Y)$ is independent of $Y$ it may seem silly to ever construct such a definition, with a 'redundant' $Y$, to replace $f$, or alternatively that such a redundant $Y$ would arise inadvertently in $g$ to be removed by replacement with the equivalent $f$. The following theorem, however, demonstrates that such 'redundant' additions can be of considerable use.
Theorem 2.1: For any recursive definitions $f$ in $F$ there is a strongly equivalent definition in $F$ which has a uniform inverse.

Proof: If $f$ already has a uniform inverse it serves as its own strongly equivalent definition. If not the following definition serves that purpose. Referring to Def. 2.1 for the definition of $f$, the following function $g$ defined in terms of the same sets, primitives and predicates is strongly equivalent to $f$. ($\rho = <\rho_1, \ldots, \rho_n>$ is a vector which records indices, and $d$ is the initial data structure.)

$$
\begin{align*}
    g(X, \rho, d) &= q(X) \quad \text{if } T(X) \\
    g(X, \rho, d) &= w(\omega_1(X), <l> /\rho, d), \ldots, g(\omega_m(X), <m(X)> /\rho, d) \quad \text{if } \overline{T}(X) \\
    \text{initially } &<X, \rho, d> = <d, \lambda, d> \text{ with } d \in D.
\end{align*}
$$

$g$ is strongly equivalent to $f$ by application of lemma 2.5. Furthermore $g$ has a uniform inverse which is given by the following:

$$
\begin{align*}
    \overset{i}{\omega}_0(X, \rho, d) &= \rho_1 \\
    \overset{0^\dagger}{\omega}(X, \rho, d) &= \omega_2 (\ldots (\omega_{n-1} (\omega_n (\rho_n (d))) ), \rho [\rho_1 \leftarrow \lambda], d)
\end{align*}
$$
The $O^{-1}$ function is quite complex, requiring recreating a sequence of data-structures starting with the initial data structure. In practice one wants to construct a strongly equivalent definition which gives an inverse but entails the creation of an $O^{-1}$ function which is simple. Simpler, hopefully, than that given in the above theorem. This can often be done. If, for example,

Corollary 2.1: For a given recursive definition $f \in F$ there is no uniform inverse, but each function $O_i \circ O$ has an inverse $O_i^{-1}$, then the definition for $g$ given above with the third component $d$ deleted from its arguments will serve with the additional benefit that an alternative simpler definition of $O^{-1}(x, p) = O_i^{-1}(x, p, d_1 + n)$ can be used.

This corollary can be applied to the 'Tower of Hanoi' definition ex.1.3. In that example, $O_i \in O$ has an inverse for $i = 1$ and 3 but does not quite have an inverse when $i = 2$:

$$O_1^{-1}(<x, y, z, n>) = <x, z, y, n+1>$$
$$O_2^{-1}(<x, y, z, n>) = <x, y, z, A> \quad \text{where A cannot be determined from} \ <x, y, z>, n$$
$$O_3^{-1}(<x, y, z, n>) = <z, y, z, n+1>$$

So first we slightly modify the definition of $f$ so there will be an inverse for $O_2$. Lemma 2.5 justifies this simple modification in which a component $s$ is added to store the quantity $A$ above when $i = 2$, and otherwise to remain equal to 0.
\[
\begin{align*}
&f'(\langle x, y, z, n, s \rangle) = \langle x, y \rangle \quad \text{if } n = 1 \\
&f'(\langle x, y, z, n, s \rangle) = f'(\langle x, z, y, n-1, s \rangle) \# f'(\langle x, y, z, 1, n \rangle) \\
&\quad \# f'(\langle z, y, x, n-1, s \rangle) \quad \text{if } n > 1 \\
&\text{initially } \langle x, y, z, n, s \rangle = (\langle 1, 2, 3, n, 0 \rangle, n \in \mathbb{N})
\end{align*}
\]

Now \( f \) is equivalent to \( \tilde{f} \) in 1, 3 and \( \alpha_i \) has an inverse for \( i = 1, 2, \) or 3.

These inverses are:

\[
\begin{align*}
\alpha_1^{-1}(\langle x, y, z, n, s \rangle) &= \langle x, z, y, n+1, 0 \rangle \\
\alpha_2^{-1}(\langle x, y, z, n, s \rangle) &= \langle x, y, z, s, 0 \rangle \\
\alpha_3^{-1}(\langle x, y, z, n, s \rangle) &= \langle x, y, z, n+1, 0 \rangle
\end{align*}
\]

Corollary 2 now applies to \( f' \). Its application yields \( g \) below.

(Some unnecessary \( >'s \) and \( <'s \) have been dropped.)

\[
\begin{align*}
g(\langle x, y, z, n, s, \rho \rangle) &= \langle x, y \rangle \quad \text{if } n = 1 \\
g(\langle x, y, z, n, s, \rho \rangle) &= g(\langle x, z, y, n-1, s, 1 \rangle \# \rho) \# g(\langle x, y, z, 1, n, 2 \rangle \# \rho) \# g(\langle z, y, x, n-1, s, 3 \rangle \# \rho) \quad \text{if } n > 1 \\
&\text{initially } \langle x, y, z, n, s, \rho \rangle = \langle 123, n, 0, \lambda \rangle
\end{align*}
\]

and the uniform inverse is given by

\[
\begin{align*}
i_0(\langle x, y, z, n, s, \rho \rangle) &= \rho_1 \\
\alpha_i^{-1}(\langle x, y, z, n, s, \rho \rangle) &= \alpha_i^{-1}(\langle x, y, z, n, s \rangle), \rho_1 [\rho_1 \cdot \lambda]
\end{align*}
\]
Implementation of $f \in F$ with Associativity and Uniform Inverse

We will give a way of implementing any $f$ in $F$ which has a uniform inverse and in which $w$ is associative. The implementation is described by a flowchart containing, as usual, interconnected assignments and decision statements. The expressions in the assignment statements and decisions are compositions involving the primitive functions and predicates $w$, $q$, $o$, $l$, $o$, $m$ and $T$ and the inverses $G$, $i_0$ which enter the definition of $f \in F$.

In addition to the above functions from the definition of $f$, the repertoire of flowchart expression is completed by an add 1 function, a push and pop and an $=$ predicate. There is a storage cell $X$ which is assumed adequate to hold any member in $\Delta_f^u Q_f V D_f$. Although there is such a push and pop, the list on which they operate, $V$, can hold at most only 2 members in $W_f^u V^w_f$.

The flowchart which follows describes a computation for each $d \in D$. It is necessary to give a concrete interpretation of the sense in which a flowchart describes a computation. We imagine a traveler who starts by entering block (0) of the flowchart. The traveler carries out the computation described in that block, then depending on the nature of the block, proceeds to the appropriate next block. The traveler continues following the block instructions and proceeding through the flowchart until FINI is reached completing the voyage. The value found in $V$ when the traveler has completed the voyage is the value computed by the flowchart.
Flowchart: notation and assumptions

In the flowchart we will use the following notation. General:

(e is an expression)

- $X \leftarrow e$: the value of $e$ is assigned to $X$
- $V \underset{\text{PUSH}}{\leftarrow} e$: the value of $e$ is pushed into list $V$
- $X \underset{\text{POP}}{\leftarrow} V$: the top member of $V$ is popped and assigned to $X$
- $X \underset{\text{POP}}{\leftarrow} V[n]$: the top $n$ members of $V$ are popped and assigned to $X$

If $V$ is a list $= \langle v_1, v_2, \ldots, v_n \rangle$ then $w(V)$ stands for the expression $w(v_1, v_2, \ldots, v_n)$.

Primitives and their Compositions: (Some of the definitions are extended to $Q_f$ to make the flowcharts work if the initial data-structure is terminal.)

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST.KID($X$)</td>
<td>$\sigma_1(X)$ if $X \in \Delta_f$</td>
</tr>
<tr>
<td>#KIDS($X$)</td>
<td>$\mu(X)$ if $X \in \Delta_f$; $\equiv 1$ if $X \in Q_f$</td>
</tr>
<tr>
<td>$X =$ TERMINAL?</td>
<td>$T(X)$ if $X \in \Delta_f \cup Q_f$</td>
</tr>
<tr>
<td>PARENT ($X$)</td>
<td>$\sigma^{-1}(X)$ if $X \in \Delta_f$; $\equiv X$ if $X \in Q_f$</td>
</tr>
<tr>
<td>SIB#($X$)</td>
<td>$i_0(X)$ if $X \in \Delta_f$; $\equiv 1$ if $X \in Q_f$</td>
</tr>
<tr>
<td>NEXT.SIB($X$)</td>
<td>$\sigma_{SIB#}(PARENT(X))$ if $X \in \Delta_f$</td>
</tr>
<tr>
<td>#SIBS($X$)</td>
<td>$#KIDS(PARENT(X))$ if $X \in \Delta_f$; $\equiv 1$ if $X \in Q_f$</td>
</tr>
</tbody>
</table>

If $w$ is associative we assume that there is a member $Q_w$ in the range of $w$ such that $w(X, Q_w) = X$ for all $X$ in the range of $w$. 
Flowchart:

For $f \in F$ and $f$ has a uniform inverse and $w$ is associative.

```
Flowchart:

For $f \in F$ and $f$ has a uniform inverse and $w$ is associative.
```

Figure 2.1
When we say a flowchart implements or realizes an $f \in F$ we mean that for each $d \in D$ the evaluation of the function $f(d)$ is to the value computed by the flowchart with traveler starting at block $\lbrack 0 \rbrack$ and $d$ in the flowchart $\lbrack 0 \rbrack$ in $f(d)$.

We now present proofs that the given flowchart

- does implement $f \in F$ under the appropriate constraints. The proof uses

induction on the remoteness of the data-structures

in $A_f$.

Theorem 2.2: If $f \in F$ and $f$ has a uniform inverse and $w$ in the definition

of $f$ is associative then $f$ is implemented by the flowchart of

(figure 2.1).

Proof: First we need to show that if $A$ of remoteness $n$, block $\lbrack 1 \rbrack$ of the

flowchart of figure 2.1 is entered with $A$ in $X$ and $B$ in $V$ then

eventually the traveler arrives at $\lbrack 3 \rbrack$ with $A$ still in

$X$ and with $V$ containing

$w[w[B, f(o_1(A))], f(o_2(A)), \ldots, f(o_m(A)(A))]$ =

by associativity

$w[B, w[f(o_1(A)), \ldots, f(o_m(A)(A))] = w[B, f(A)]$

Again we use induction. The case when $A$ is of remoteness 1

is easily verifies by tracing the flowchart through the

sequence of blocks $\langle \lbrack 1 \rbrack \lbrack 3 \rbrack \lbrack 1 \rbrack \lbrack 2 \rbrack \lbrack 6 \rbrack \lbrack 4 \rbrack \lbrack 5 \rbrack \rangle \cdot m(A)-1$ times and

then through $\lbrack 1 \rbrack \lbrack 3 \rbrack \lbrack 1 \rbrack \lbrack 2 \rbrack \lbrack 6 \rbrack \lbrack 4 \rbrack \lbrack 7 \rbrack \lbrack 9 \rbrack$. 
Assume **First** is correct if the remoteness of A is < n. Now let A be of remoteness n; X is A, V is B and the traveler is at ①. The traveler goes to ② where X becomes **FIRST**, KID(X) = o₁(A) and the traveler returns to ①. Since o₁(A) is of remoteness < n, the inductive hypothesis applies. Thus the traveler arrives at ⑨ with X being o₁(A) and V = w[B, f(o₁(X))].

o₁(A) cannot be initial because of the inverse so the traveler goes next to ④. If we assume now that 1 = SIB#(X) ≠ #SIBS(X) where X = o₁(A), the traveler will pass through ④ and ⑤ updating X to contain o₂(A) and then enter ①. By inductive hypothesis again the traveler will eventually arrive at ⑨ with V containing:

\[ w[B, f(o₁(A))], f(o₂(A)) \]

and X containing o₂(A). Assuming without loss in generality that p = m(A), the traveler will eventually arrive at ⑨ after p repeats of the journey from ① to ⑨ with V containing:

\[ w[w[B, f(o₁(A))], f(o₂(A))], \ldots, f(o_p(A))] = w[B, w[f(o₁(A)), \ldots, f(o_p(A))]] \]

by associativity and = w[B, f(A)] by definition of f(A).

X contains o_p(A) at this time. So the traveler goes to ④ where the decision is yes; ⑦ is next with X becoming its PARENT, i.e. PARENT(o_p(A)) = A. Thus the traveler arrives at ⑨ again with X containing A, V still containing w[B, f(A)]. Thus the First result is proven. Now let the
traveler start by entering (1) thus setting X to d and V to B = 0_w. Next the traveler enters (2) with these values in X and V and so by the First result the traveler will eventually arrive at (3) with X containing d and V containing w[0_w,f(d)] = f(d) by definition of 0_w.

As before the proof is for d ∈ D having remoteness ≥ 1 and is verified to include remoteness 0 by tracing the flowchart explicitly for this case.

The necessity for a 'uniform inverse' as opposed to a simple inverse in developing this theorem results from the fact that in the standard form of recursive definition considered here the number of appearances of the defined function symbol f is determined (=m(X)) by X the argument of f. This dependence was incorporated so that many common problems could be naturally expressed in that form.

We have not discussed the higher order recursive definitions having nesting on the right - largely because in our experience such definitions rarely occurred in practice. Such definitions are considered in [6]. The techniques given in [6] in combination with those here can be used to extend the above results to higher order recursive definitions not covered in [6].
Application of Theorem 2.2

Before applying theorem 2.5, it will be useful to make some notes on flowchart of figure 2.1.

Flowchart Notes:

1) In general, X has a number of components; these components will be saved in identified storage locations.
   In any assignment to X in the flowchart, only those locations holding components which are changed need be assigned.
   In any decision on X in the flowchart, only those component locations need be tested which are necessary to secure the decision.

2) Some of the functions in the boxes like NEXT: SIB(X) in \( \text{⑤} \), and the decision in \( \text{④} \) are sufficiently complex compositions so that simplifications are often possible.

3) The assignment in box \( \text{⑦} \) is made when data-structure X has the property that \( \text{SIB#(X)} = \#\text{SIB}(X) \).

4) Conversely, the assignment in box \( \text{⑤} \) is made when \( \text{SIB#(X)} \neq \#\text{SIB}(X) \).

Beyond these generally applicable simplifications, others are applicable when the primitive functions in the flowchart have special properties.

The following is important for our example.

If w is a union and each q(X) produced in \( \text{②} \) will be different from all others produced there, then boxes \( \text{②} \) and \( \text{③} \) may be replaced by one having the assignment OUTPUT = q(X), meaning place q(X) in the next location in an internal storage table or external (paper) table.
Theorem 2.2 applies to many enumeration problems. It produces a 'good' algorithm from an easily justified recursive definition. For example, the theorem applies to Ex 1.1, a recursive definition for enumerating binary numbers. The definition in Ex 1.1 does have an inverse - namely:

\[
\begin{align*}
\text{if } \alpha &= \langle a, \ldots, \alpha_p \rangle \\
\text{then } O^{-1}(\alpha, n) &= \langle \alpha[p] + \lambda, n+1 \rangle \\
\text{and } i_0(\alpha, n) &= \alpha_p + 1
\end{align*}
\]

Using this inverse and the \( o,w,m,T \) of ex 1.1, we get the following flowchart expression definitions:

\[
X = \langle a,n \rangle
\]

\[
\text{FIRST.KID}(\langle a,n \rangle) = \langle a//0>, n+1 \rangle
\]

\[
\text{#KIDS}(\langle a,n \rangle) = 2
\]

\[
\langle a,n \rangle = \text{TERMINAL} = n=0?
\]

\[
\text{PARENT}(\langle a,n \rangle) = \langle a[p] + \lambda, n+1 \rangle
\]

\[
\text{SIB#}(\langle a,n \rangle) = \frac{\alpha_p + 1}{p}
\]

\[
\text{SIBS}(\langle a,n \rangle) = 2
\]

\[
\text{SIB#}(\langle a,n \rangle) = \text{SIBS}(\langle a,n \rangle) = \alpha_p + 1 = 2? \quad \text{referring to note 2 we can simplify in this case; } \alpha_p = 1
\]

\[
\text{NEXT.SIB}(\langle a,n \rangle) = \langle a[p] + \lambda, n, a> \quad \text{if } a_p = 0, \text{ which is all it can equal so simplifying:}
\]

\[
= \langle a[p] + \lambda, n \rangle
\]

also observe, referring to note 1 that \( n \) is not changed.

The flowchart of figure 2 is obtained by inserting these definitions in that of figure 1 as modified according to note 5 which is valid in this case. It is essentially the 'add-one' algorithms described in the introduction.
Similarly, theorem 2.5 can be applied to ex 2.1, the permutation definition. This definition also has an inverse, namely:

\[
\text{if } \alpha = \langle a_1, \ldots, a_p \rangle, \text{ and } x = \text{the position (index) of the integer (n+1) within } \alpha
\]

\[
\alpha^{-1}(n, a) = \langle n+1, a[\alpha_x + 1] \rangle
\]

\[
i_0(n, a) = x
\]

Most of the flowchart definitions result from straightforward substitution of the inverse and the given primitive functions. For example:

\[
\text{SIB}\langle n, \alpha \rangle = x
\]

\[
\text{SIB}\langle n, \alpha \rangle = \text{KIDS(PARENT}(n, \alpha)) = m(n+1, a[\alpha_x + 1]) = |a[\alpha_x + 1]| + 1
\]

\[
= |a| + 1
\]
From which the decision of box 4:

\[ \text{SIB}(\langle n, \alpha \rangle) = \#\text{SIBS}(\langle n, \alpha \rangle) \text{ ?} \]  
\[ \implies \{ x = |\alpha| \} \text{ which by the interpretation of } x \text{ is equivalent to:} \]
\[ = (\alpha_p = (n+1)?) \]

Also note that:

\[ \text{NEXT.SIB}(\langle n, \alpha \rangle) = \alpha_{x+1} \langle n+1, \alpha[\alpha_x + \lambda] \rangle \]
\[ = \langle n, (\alpha[\alpha_x + \lambda])[\alpha_{x+1} + n+1] \rangle \text{ which again, by the interpretation of } x \text{ is:} \]
\[ = \langle n, \alpha[\alpha_x, \alpha_{x+1} - \alpha_{x+1} \alpha_x] \rangle \text{ representing an interchange of the } (n+1) \text{ component in with its right neighbor} \]

These definitions when put in figure 1 as follows produce an algorithm which stores only one permutation. The next permutation is always produced by interchanging components of the currently stored one. The resultant flowchart is:

![Flowchart](image-url)

**figure 3**
Note that this flowchart contains the assignment: of the 'position of (n+1)
in a' to the variable x implying a search for the integer (n+1) in a. A
modification in the argument of the recursive definition can be made which
will obviate this search. If we add to the argument a vector β giving the
position of (n+1) in a, the 'position of (n+1)' will be available. Thus
modifying Ex 1.2:

\[ x = \{<n,\alpha,\beta> | n \in \mathbb{N}, \alpha \text{ and } \beta \text{ strings of positive integers} \} \]

\[ f(<n,\alpha,\beta>) = \begin{cases} \alpha & \text{if } n = 0 \\ f(<n-1,\alpha[\alpha_{0+} n],\beta>) & \text{if } n > 0 \end{cases} \]

x is initially \( \in \{<n,\lambda,\lambda> | n \in \mathbb{N} \} \)

The modification still has an inverse and can be implemented by the
inclusion of a variable β and following the prescription of theorem 2.5 to
obtain the following assignments to that variable in figure 2.1.

\[ \beta + \lambda \quad \text{in (1)} \]

\[ \beta + <1>/\beta \quad \text{in (2)} \]

\[ \beta + \beta[\beta_1 + \lambda] \quad \text{in (3)} \]

replace x by \( \beta_1 \) in (4) and after the assignment to a put:

\[ \beta + \beta[\beta_1 < \beta_1 + 1] \]

This illustrates how with this approach one can modify an algorithm
focusing on the effect of a change in data-structure without ever having to
be enmeshed in the control structure of the algorithm.

The Search for the Inverse

We have been building a system which examines a recursive definition
for properties identified in this paper as sufficient to allow efficient
implementation. An important part of this system is a program for determining whether the given definition has a uniform inverse. For a recursive definition of the standard form:

let the data-structure $X$ be a vector with $n$ components, i.e.

let $X = <x_1, \ldots, x_n>$

Let the primitive $O$-functions be:

\[ o_1(X) = o(1, X) = o(1, x_1, \ldots, x_n) = <y_1, \ldots, y_n> \]

Let $o[j](I, X)$ be the $j^{th}$ component of $o(I, X)$, thus:

$\begin{align*}
    y_1 &= o[1](I, x_1, \ldots, x_n) \\
    y_2 &= o[2](I, x_1, \ldots, x_n) \\
    &\vdots \\
    y_n &= o[n](I, x_1, \ldots, x_n)
\end{align*}$

It is easy to see that the recursive definition has a uniform inverse iff there is a unique solution of set $A$ for $I$ and each $x_j$ as functions of $y_1$ through $y_n$. If the solution for $I$ is $r(y_1, \ldots, y_n)$ and for $x_j = t_j(x_1, \ldots, x_n)$, then the uniform $i_0$ and $O^{-1}$ functions in the uniform inverse are given by:

\[ i_0(y_1, \ldots, y_n) = r(y_1, \ldots, y_n) \]
\[ O^{-1}(y_1, \ldots, y_n) = t_1(y_1, \ldots, y_n), t_2(y_1, \ldots, y_n), \ldots, t_n(y_1, \ldots, y_n) \]

Currently we have implemented a search for a 'simple' uniform inverse. This is described below.

The first step is to set up the equations of set $A$ for a given recursive
definition and then to try to obtain a solution by the following simple procedure.

1. First each equation in the set is tested to determine whether that equation by itself can be inverted. Often the right hand side of the equation will be only dependent on some of the variables $x_1, \ldots, x_n$ and $I$, and such an inversion will be possible. For example, say $y$ on the left only depends on one $x$ on the right:

   \[
   \begin{align*}
   \text{if} \quad y &= x + 1 \quad \text{then} \quad x &= y - 1, \quad \text{again assuming $y$ is a vector then} \\
   \text{if} \quad y &= \langle 0 > / x \quad \text{then} \quad x &= y[\gamma_1 + \lambda], \quad \text{also} \\
   \text{if} \quad y &= I \not\parallel x \quad \text{then} \quad \begin{cases} x &= y[\gamma_1 + \lambda] \\ I &= y \end{cases} \\
   \text{if} \quad y &= \langle 0 > / x \quad \text{if $l = 1$} \\
   \quad \text{if $l = 2$} & \quad \begin{cases} x &= y[\gamma_1 + \lambda] \\ I &= \{ 1 \text{ if $x = 0$} \\ 2 \text{ if $y = 1$} \end{cases} \\
   \end{align*}
   \]

2. If there is no simple equation which can thus be inverted, then the conclusion is that a 'simple' uniform inverse is not available. This does not mean there is no uniform inverse, but since the computation of the inverse when it exists will become part of the implementing flowchart there is some justification in limiting our search to uniform inverses which are relatively simple to compute.

   If one or more equations can be inverted, obtaining solutions for some $x_j$'s and perhaps for $I$, the solutions are substituted for those $x_j$'s and perhaps $I$ in the other unsolved equations and the procedure returns to step 1,
applying it to the unsolved equations with their substitutions.

If finally all equations are solved, the 'simple' uniform inverse has been obtained.

Simple as this procedure appears there is still considerable difficulty in determining how and if a simple equation can be inverted when one is dealing with relatively exotic functions such as concatenation, decision, insertion, etc. which arise in actual recursive definitions. At this stage even this simple step is handled heuristically for a limited number of primitive functions with no guarantee that inverses will always be produced when they exist.
References:

1. Aho, Hopcroft, Ullman; The Design and Analysis of Computer Algorithms; Addison-Wesley, 1975; pp 195-222

2. Darlington, J. and Burstall, R.M.; A System Which Automatically Improves Programs; Proceedings Third International Joint Conference on Artificial Intelligence; Stanford, California, 1973; pp 479-485


7. Paull, M.C.; Formulation and Manipulation of Enumeration Based Algorithms; Research Report SOSAP-TR-4; December, 1973

Appendix I

Summary of Frequently used Notation

If \( P \) is a predicate the \( \overline{P} \) means not \( P \).

\( \mathcal{N} \) is the set of all positive integers \( = \{1,2,\ldots\} \)

\( \mathcal{N} \) is the finite set of integers from 1 to \( n \) \( = \{1,\ldots,n\} \)

If \( A \) and \( B \) are sets

\( A \cup B \) is set union

\( A \cap B \) is set intersection

\( \overline{A} \) is the complement of \( A \)

\( A - B = A \cap \overline{B} \)

\( |A| \) = the number of elements in \( A \)

\( \langle a_1, \ldots, a_n \rangle \) is an ordered set or vector with components

\( a_i : i \in \mathcal{N} \) and \( a_{i:j} \) represents the subvector \( \langle a_1, a_{i+1}, \ldots, a_j \rangle \); \( a_{1:1} = \langle a_1 \rangle \)

If \( A \) and \( B \) are ordered sets \( = \langle a_1, \ldots, a_n \rangle \) and \( \langle b_1, \ldots, b_n \rangle \) respectively

\( A/B = \langle a_1, \ldots, a_n, b_1, \ldots, b_n \rangle \)

\( \{A\} \) is the set of all components in \( A \)

If \( E \), \( x \) and \( y \) are each an expression, i.e. a string or ordered set of symbols from a given alphabet, usually satisfying some constraints as to form, then

\( E[x \leftrightarrow y] \) is the expression that results when each occurrence of \( x \) is replaced by \( y \) in \( E \).

The notation is extended to allow the specification of a number of replacements \( E[x \leftrightarrow y, Z \leftrightarrow w] \) is the expression which results when each \( x \) is replaced by \( y \) and each \( Z \) by \( w \) in \( E \).