RELATIONS BETWEEN RECURSIVE DEFINITIONS AND THEIR EFFICIENT IMPLEMENTATIONS.

BY: M.C. Paul
January 1977
DCS-TR-#57

Dept. of Computer Science
Hill Center, Busch Campus at
Rutgers, The State Univ. of N.J.
New Brunswick, N.J. 08903

This work was supported by Grant #DAHC15-73-G-6
The Advanced Research Projects Agency.
RELATIONS BETWEEN RECURSIVE DEFINITIONS
AND THEIR 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 digraph 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 one of the main subjects of this paper.

Implementation of the recursive definition generally requires the repetitive execution of similar operations. If it can be shown that some pairs of these operations will yield the same or similar intermediate results at different points in the computation--then only one such intermediate result need be computed and remembered. It may then be accessed from memory when needed again instead of being recomputed. This can happen many times in a sufficiently systematic way so that a significant time saving can be realized. The existence of this situation depends on properties of the primitive functions which compose the recursive definition. A significant class of problems for which time efficient implementations are available are explored in this paper.
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
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 O-functions had an inverse - in example 1.5 the O-function as given does not have an inverse and thus the
algorithm scheme $S(V')$ is not available. However, as will be shown* - when in a recursive definition, $f \circ f(V)$, the $O$-function does not have an inverse - a simple transformation of $f$ to an equivalent definition, $f'$, involving an $O$-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 $f'(d') = f(d)$. 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 $O$-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

* Theorem 2.3.
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 o_i' than the corresponding X and o_i of f. That is, for each
definition of form I there is an equivalent definition
of the form:*  

\[
\begin{cases}
  f'(X') = q'(X') & \text{if } T(X') \\
  f'(X') = w(f'(o'(X'))) & \text{if } T(X') \\
  \text{Initially, } X' \in D_f,
\end{cases}
\]

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:

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

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

* Theorem 2.1 and theorem 2.2 give the two classical ways this is done,
called breadth-first and depth-first respectively.
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.

**Time Efficient Implementations**

As noted earlier the opportunity for time efficient implementations of recursive definitions in $F(V)$ (= $F$ from here on) arises when repetitive use of the same operations are necessary in the evaluation of the function. In the time efficient implementations the originally repeated operations are done once - the result being remembered for later use. This is classically called 'pruning' in the Artificial Intelligence literature. For some subclasses of $F$ the nature of the repeated operations are sufficiently independent of the particular initial data-structure so that one can design a class of implementations guaranteed to be more efficient in all cases than the standard implementation. An example of such a subclass is all functions which are substitutionally solvable and are of the form given below.

Let $\vec{V} = \langle x_1, x_2, \ldots, x_n \rangle$ = a vector whose values are integers, 
$\vec{C}_i = \langle c_{i1}, \ldots, c_{in} \rangle$ = a constant vector of integers - is vector subtraction.

\[
\begin{cases}
    f(\vec{V}) = q(\vec{V}) & \text{if } \vec{V} \text{ is any of a finite set of integer vectors, say } K \\
    f(\vec{V}) = w(f(\vec{V}-\vec{C}_1), \ldots, f(\vec{V}-\vec{C}_p)) & \text{if } \vec{V} \notin K \\
    \text{Initially } \vec{V} \in \text{ a defined set of integer vectors, say } L \\
\end{cases}
\]

A specific simple member of this subclass is the definition for the
Fibonacci series ($\mathbf{V}$ is a one dimensional vector)

\[
\begin{align*}
  f(n) &= 1 & \text{if } n = 1, \text{ or } 0 \\
  f(n) &= f(n-1) + f(n-2) & \text{if } n > 1 \\
  \text{Initially } n \in N
\end{align*}
\]

For any definition in this subclass in which $\mathbf{V}$ has $m$ components it is easy to see that the function can be implemented by $m$ nested DO-loops.

This implementation requires much less than the exponential time involved in a straightforward implementation of the recursive definition - which does not take advantage of repeated suboperations. Although detection of a member of this subclass is relatively easy, the detailed dependence of the parameters of the nested DO implementation on the properties of $\mathbf{w}$ and the and the sets $K$ and $L$ in this subclass of definition is an interesting study. This study however is not carried out here.

Another such subclass called the 'explicit history' class will be considered here. The evaluation of members of this class will be shown to be equivalent to solving a set of equations* analogous to sets of linear equations. Consequently an algorithm analogous to Gaussian elimination will be shown to be an available implementation of explicit history definitions. This algorithm has polynomial complexity as opposed to the exponential time required in a straightforward implementation of the recursive definitions in this class.

* Theorem 3.1.
Equivalent Formulations of Recursive Definitions

To be in the class \( F(V) \) recursive definitions must be of form I.

Despite this constraint a function may have a number of different
definitions all in \( F(V) \). Some of these may have desirable properties
absent in others. Transformations which can be used to obtain equivalent
definitions in \( F(V) \) with desirable properties are developed in section 3
of this paper. Actually, example 1.1 gives a definition of the set of
all \( n \)-bit binary numbers which may not be entirely natural. One which
may be considered is based on the fact that:

The set of all \( n \)-bit binary numbers = the set of all \( (n-1) \)-bit
binary numbers each with a 0 appended together (united) with the set of
all \( (n-1) \)-bit binary numbers each with a 1 appended.

A formal statement of this definition is:

Ex. 1.5 \[
\begin{align*}
\text{if } n = 0 & \quad f(n) = \{ \lambda \} \\
\text{if } n > 0 & \quad f(n) = (0 \circ \bigoplus f(n-1)) \cup (1 \circ \bigoplus f(n-1))
\end{align*}
\]

where \( B \) is a set of strings and \( \alpha \) a string \( \alpha \bigoplus B = \{ \alpha \cup b | b \in B \} \)

Initially \( n \in N \)

Then ex. \( f(2) = \{ \alpha \circ \bigoplus f(1) \cup (1 \circ \bigoplus f(1)) \}
\]

This definition, though still strictly in form I is not in \( F(V') \)
because \( \circ \) is not associative. Therefore the scheme \( S(V') \) is not directly
available for its realization. However, there are theorems* which will

* Theorem 3.2
in this case give an equivalent definition in \( \Gamma(V') \) thus once again making
the realization of \( \Sigma(V') \) available.

In fact, the definition of example 1.1 can be obtained by the application
of such theorems to example 1.5. It is interesting to compare the above
interpretation of the definition in example 1.5 with one for the equivalent
definition in example 1.1 which was originally claimed to be natural.

Interpretation of Definition 1.1:

The set of all \( n + |\alpha| \)-bit binary numbers which have a prefix \( \alpha = \)
The set of all \( n + |\alpha| \)-bit binary numbers having a prefix \( \alpha \) followed
by 0 together (unioned) with the set of all \( n + |\alpha| \)-bit binary numbers
having a prefix \( \alpha \) followed by 1.
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

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

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 $X \in \Delta_f$.

$O$ 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 $O$-function.

The range of $o_1$ must include the set $p_i$.

The union of the sets $p_i$ of all the functions in $O$ is the range of the $O$-function and is called $P_f$.

$T$ is a predicate whose domain includes $D_f \cup P_f$. Its range is \{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 \( W_f \cup W_{\bar{f}} \).

The sets named above are defined as follows (the subscript \( f \) is dropped where it is not essential):

\[ \Delta^1 = \{ d \in D : d \in T(d) \} \quad \text{and for } j > 1 \]

\[ \Delta^j = \{ o_i(x) | x \in \Delta^{j-1} \text{ and } i \leq m(x) \text{ and } T(o_i(x)) \} \]

\[ \Delta = \bigcup_{i=1}^{\infty} \Delta^i \]

The set \( \delta^i \) of \( o_i \in O \) is:

\[ \delta^1 = \{ x | x \in \Delta \text{ and } i \leq m(x) \} \]

The range of \( O \) is:

\[ P = \{ o_i(x) | x \in \Delta \text{ and } i \leq m(x) \} \]

The range \( p^i \) of \( o_i \in O \) is:

\[ p^i = \{ o_i(x) | x \in \delta^i \} \]

The set of terminal data-structures \( Q \) is:

\[ Q = P - \Delta \]

The set \( W \) is defined as follows:

\[ W^1 = \{ w(x_1, \ldots, x_n) | x_i \in w_f, n = \text{a positive integer } N \} \quad \text{for } j > 1, \]

\[ W^j = \{ w(x_1, \ldots, x_n) | x_i \in W^k, k(j); n \in N \} \]

\[ W = \bigcup_{i=1}^{\infty} W^i \]

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 \( \langle i_1, ..., i_n \rangle = \text{I} \) is a sequence of integers then \( o_{\langle i_1, ..., i_n \rangle}(x) = o_{i_1}(x) \) is an abbreviation for \( o_{i_2}(o_{i_1}(x)) \) \( \ldots \); \( o_{\lambda}(x) = x \).
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 \(\forall d \in D\), \(d \in D\) implies \(I(d)\) 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 \(F\). 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 \(F \in F\)**

**Lemma 2.1:** If \(f \in F\) and \(X \in \Delta_f\), then there exists an integer sequence \(I \in I(d)\) and a data-structure \(d \in D\) such that \(o_{<i}> (d) = X\).

**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 \(\Delta^i\). 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 \(o_{<i>_Y} (d) = Y\). Then it follows, by definition of \(\Delta^k\) that if \(X \in \Delta^k\) then \(X = o_{<i>_Y} (d)\) for some \(i \leq m(Y)\), and \(Y \in \Delta^{k-1}\). Thus \(X = o_{<i>_Y} (d)\) for \(i \leq m(Y)\), and \(o_{<i>_Y} (d) = d\) for each \(d \in D\), the proof is complete.

**Lemma 2.2:** If \(F \in F\) and \(X \in \Delta_f\), then \(I(X)\) is finite.

**Proof:** From the previous lemma the data-structure \(X = o_{<i>_Y} (d)\) for some \(d \in D\) and integer sequence \(I\). Therefore \(I(d) = \text{the}\)
set consisting of I 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 ∈ F is substitutionally solvable.

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

Ordering the Data-Structures in ∆ (Remoteness):

For any function f in F:

We say a data-structure X in ∆_f ∪ Q_f is of remoteness 0 (or is terminal) if X ∈ Q_f.

We say a data-structure X in ∆_f ∪ Q_f is of remoteness n if:

(1) ∃ i: i ≤ m(X) and o_i(X) is of remoteness n-1 and

(2) ∀ i: i ≤ m(X) implies o_i(X) is of remoteness n-k and kal.*

Lemma 2.3: If f ∈ F, then there is a function r with domain ∆_f ∪ Q_f such that if X ∈ ∆_f ∪ Q_f then r(X) = the remoteness of X.

Proof: For each X ∈ ∆_f ∪ Q_f let r(X) be the maximum of the length of all the sequences in I(X). For each X ∈ ∆_f ∪ Q_f, X is of

* 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 \( I = <i_1, \ldots, i_n> \) in \( I(X) \). Let
\( o_i(X) = Y \). Then \( I' = <i_2, \ldots, i_n> \) is in \( I(Y) \). Furthermore,
no sequence applicable to \( Y \) is longer than \( I' \) because other-
wise \( I \) could not have been a longest sequence in \( I(X) \). So
\( r(Y) = n-1 \) and \( Y \) is of remoteness \( r(Y) = n-1 \). Therefore,
since \( o_i(X) = Y \) and for all \( j \neq i_1, j \leq m(X), r(o_j(Y)) \leq n-1 \),
\( X \) is of remoteness \( r(X) = n \) by definition of remoteness.

Interpretation of the Recursive definitions in \( F \)

In the next paragraphs we briefly sketch some important well known
facts about the interpretation of substitutionally solvable recursive
definitions such as \( f \in F \).

A recursive definition \( f \in F \) defines the function \( f \) on the domain
\( \Omega_f \) by giving a relation (in terms of the primitive function \( w \)) that \( f(d) \)
must satisfy with the same function \( f \) at some different argument value
namely with \( f(o_i(d))'s \) for \( 1 \leq i \leq m(d) \). The same definition is
applicable to define \( f(o_i(d)) \) for each \( 1 \leq i \leq m(d) \),
of \( f \) at still other arguments. This process of repeated redefinition of
\( f \) with different arguments will eventually end (because \( f \in F \) is sub-
stitutionally solvable) with arguments \( X \) which are terminal, at which point
the definition of \( f \) gives a definite value \( q(X) \) to be assigned to \( f(X) \).
Thus this process will close, and a definite value will be assigned to
\( f(d) \). It can easily be shown that this is a unique value.
This process of re-definition can be formulated as a non-deterministic procedure involving successive substitutions in an expression whose evaluation will give the value of \( f(d) \) for \( d \in D_f \).

Let \( E_i \) be an expression involving a composition of the \( w \), and \( o_i \) functions, \( a, d \in D \), and occurrences of \( f(a) \), \( f \) being the symbol for the defined function, and \( a \) its argument. Let \( E_1 = f(d) \) and in general to get \( E_{i+1} \) from \( E_i \) do the following:

Choose any occurrence of \( f(a) \) in \( E_i \). Note that \( a \) itself will never contain any occurrence of \( f \), \( a \) will just involve a composition of \( w \)'s, \( o_i \)'s and \( d \). Evaluate \( a \), this can be done because it only involves given primitive functions and a given data-structure \( d \).

If the evaluated \( a \) is terminal, i.e. \( T(a) \) is true, then \( f(a) \) is replaced by \( q(a) \) to obtain \( E_{i+1} \). Thus the right side of the 1st equation of the definition of \( f \) is substituted for \( f(a) \). If on the otherhand \( T(a) \) then the value of \( a \) is substituted for \( X \) on the right side of the second equation in the definition of \( f \), and then this entire resulting right side is substituted for \( f(a) \) in \( E_i \) to produce \( E_{i+1} \).

Substitutions are continued until \( E_i \) contains no occurrences of \( f \). This must eventually occur because \( f \) is substitutionally solvable. At this point in the evaluation \( E_i \) is the value of \( f(d) \).

The result of this non-deterministic procedure starting with \( f(d) \) may be interpreted as the definition of \( f(d) \).

This definition is non-deterministic because any occurrence of \( f(a) \) in \( E_i \) may be legitimately chosen to be substituted for next, no order is prescribed.
Although the meaning of the recursive definition is tied to this non-deterministic unordered procedure, the common connotation of 'recursive implementation' involves a fixed ordering of the substitutions for occurrences of $f(a)$ in the successive expressions $E_i$. This order requires substitution always for the leftmost occurrence of $f(a)$ in $E_i$. This is the order implemented in virtually all compilers which accept recursive definitions. It is sometimes called depth-first ordering. This ordering amongst others will be investigated here. We call the depth-first ordering the standard implementation--recognizing that strictly there is not a single implementation entitled to be called the recursive implementation.

So given a recursive definition--and the order in which the $f(a)$ occurrences are to be substituted for--the basis of a deterministic implementation is established. This can be detailed in a flowchart and is one of the ways in which we will specify such an implementation.

There is a subset of recursive definitions of $f \in F$, however, in which one need not specify the order explicitly. There can be no question of the order of substitution for occurrences of $f(a)$ when each expression $E_i$ has only one occurrence of $f(a)$. This will occur for any definition of $f \in F$ whose second equation has only one occurrence of $f(a)$ on the right. Such a unary recursive definition itself then is a second way to specify an implementation.

An implementation which can be specified in one of those forms can also be specified in the other form.

In the subsequent sections both ways of specifying an implementation are used.
In the next section unary definitions are developed which represent the classical depth-first and breadth-first implementations of functions \( f \in F \). By showing the equivalence of these unary definitions to \( f \in F \), the validity of these implementations is established.

Later another implementation for \( f \in F \) called an inverse implementation will be described by a flowchart and will be proven to be valid.

**Classical Implementations:**

Notation:

We need notation for operations which replace a component of a vector with single or multiple components which are functions of the replaced component.

Let \( L \) be a vector (list); \( L = \langle l_1, \ldots, l_n \rangle \).

Let \( t_i \) denote an individual member or subsequence of \( L \) which has some specified properties \( P_i \). The notation:

\[
L[t_1 \rightarrow X_1; \ldots; t_n \rightarrow X_n]
\]

denotes the list obtained by replacing all components of \( L \) which have property \( P_i \) by \( X_i \). \( X_i \) may be an individual component or a number of components. For example, if \( P_1 \) is the property of being an odd indexed component of \( L \) and if \( n (= \text{the number of components of } L) \) is even then the meaning of \( L[t_1 \rightarrow a] \) is given by:

\[
L[t_1 \rightarrow a] = \langle a, l_2, a, l_4, a, \ldots, l_n \rangle
\]

The next two theorems will show that for each \( f \in F \) there are two unary definitions \( F_B \) and \( F_D \) both also members of \( F \) and both equivalent to \( f \). Since they are unary the implementation

---

*Depth-first and Breadth-first algorithms are described in [4] as algorithms for searching a state space. Such searches (and more) can usually be modelled by a recursive definition of our standard form recursive definition - with the nature of the transition from state to state given by the \( \theta \)-function and the nature of the search given by the \( w \)-function.*
for evaluating these definitions is deterministic (see pg 19). The
algorithm $F_B$ is similar to the classical 'breadth-first' algorithm and
$F_D$ is similar to the classical 'depth-first' algorithm.

Breadth-First Implementation

Let $f \in F$, thus

1) \[ f(X) = q(X) \quad \text{if } T(X) \]
2) \[ f(X) = w(f(o_1(X)), \ldots, f(o_m(X))))) \quad \text{if } \overline{T}(X) \]
3) \[ \text{initially } X \in \mathcal{D} \]

To define the function $F_B$ which is equivalent to $f$ we first need to define
a number of new primitive functions and predicates in terms of the
primitive functions $Q_w, q, m$ and predicate $T$ of $f$. For this the notation
just introduced is used.

$L = \langle l_1, \ldots, l_n \rangle, \ Z = \langle z_1, \ldots, z_p \rangle$ are both vectors. The components
of $L$ are either brackets in the set

\[ \text{BRACK} = \{ '(', ')', '<', '>', '{', '}' \}, \]

or members of $\Delta_f = \text{the domain of } f$, or of $Q_f = \text{the terminal data-structures}$
$\text{of } f$. ($\Delta_f$ and $Q_f$ are assumed not to contain any of the brackets in BRACK.)
The components of $Z$ are either members of BRACK or of $w_f = \text{the range of } q$,
or of $w_f = \text{the range of } w$.

With $X \in \Delta_f$, $t \in Q_f$, and $t_i \in w_f \cup w_f$, let:

\[ Q_B(L) = L[X + '{', o_1(X), \ldots, o_m(X), '}'] \]
\[ T_B(L) = \text{true if every component of } L \text{ is a member of BRACK or of } Q_f \]
\[ Q_B(L) = L[t + q(t)] \]
\[ M_B(L) = 1 \]
\[ w_B(Z) = Z[ '{', t_1, \ldots, t_n, '}'] + w(t_1, \ldots, t_n) \]
Now we can define $F_B$:

II 1) $F_B(L) = Q_B(L)$ if $T_B(L)$

2) $F_B(L) = W_B(F_B(Q_B(L)))$ if $\overline{F}_B(L)$

3) Initially $L = \langle X \rangle$, $X \in D$

Theorem 2.1: For each $f \in F$ (as I above) there is a function $F_B \in F$ (as II above) such that $F_B(\langle X \rangle) = \langle f(X) \rangle$ for all $X \in D$.

Proof: The proof uses induction on the remoteness of the data-structures $X \in \Delta_f \cup Q_f$ which, along with brackets, constitute the significant components of the vector data-structures $L \in \Delta_{F_B}$.

What we will show is first that for any $L$ in $\Delta_{F_B}$ whose components are all members of $Q_f$, designated by $t$, or are in $\text{BRACK}$, that:

$$F_B(L) = L[t \cdot f(t)]$$

This is true since if $T_B(L)$ is true then with $t \in Q_f$

$$F_B(L) = Q_B(L) \quad \text{by II 1}$$

$$= L[t \cdot q(t)] \quad \text{by definition of } Q_B$$

$$= L[t \cdot f(t)] \quad \text{by I 1}$$

Secondly we show inductively that if $L$ contains component of $\Delta_f$ designated $X$ then

III $F_B(L) = L[X \cdot f(X)]$

Assume that as long as all components of $L$, other than those in $\text{BRACK}$, are of remoteness $\leq n$ that statement (III) is true. If $T_B(L)$ is not true and all components of $L$ other than those in $\text{BRACK}$ are of remoteness $\leq n$, $n > 0$, and at least one such component is of remoteness $n$, then: $(X$ is used to designate a member of $\Delta_f$, $t$ a member of $Q_f)$
-25-

\[ F_B(L) = W_B(F_B(O_B(L))) \text{ by II(I)} \]
\[ F_B(L) = W_B(F_B(L[X = \{', o_1(X), \ldots, o_m(X)(X), '\}']) \]

by the definition of \( O_B \)
\[ F_B(L) = W_B(F_B(L')) \text{ abbreviating the expression above with } L' \]

Clearly all components in \( L' \) are of remoteness \(< n \) and at least one has remoteness \( n-1 \).

So the inductive hypotheses may be used for all \( X \in \Delta_f \) in \( L' \).
\[ F_B(L) = W_B(L'[X = f(X)]) \]

but \( L' = L[X = \{', o_1(X), \ldots, o_m(X)(X), '\}'] \)

So \( L'[X + f(X)] = L[X + \{', f(o_1(X)), \ldots, f(o_m(X)(X)), '\}'] \)

So \( W_B(L'[X + f(X)]) = L[X + w(f(o_1(X)), \ldots, f(o_m(X)(X)))]) \)

by the definition of \( W_B \).

Thus:
\[ F_B(L) = L[X + f(X)] \]

Thus for \( L = \langle X \rangle, X \in D \):
\[ F_B(L) = \langle X \rangle[X + f(X)] = <f(X)> \]

Depth-First Implementation

The depth-first function \( F_D \) equivalent to \( f \) in I above is defined as follows:

III 1) \[ F_D(L, k) = O_D(L, k) \text{ if } T_D(L) \]
2) \[ F_D(L, k) = F_D(O_D(L, k)) \text{ if } T_D(L) \]
3) \[ \text{initially } L = \langle X \rangle, X \in D_f, k=1 \text{ (we say then } \langle L, k \rangle \in D_D) \]

where with \( l_k \) the \( k \)th component of \( L \) the definition of \( O_D, T_D \) and \( Q_D \) are:

1) \[ Q_D(L,k) = \langle L[l_k \leftarrow '\{', o_1(l_k), \ldots, o_m(l_k)(l_k)'\}], k \rangle \text{ if } l_k \in \Delta_f \]
2) \[ = \langle L[l_k \leftarrow q(l_k)], k+1 \rangle \text{ if } l_k \in Q_f \]
3) \[ = \langle L[\{', t_1, \ldots, t_n', '\}'] + W(t_1, \ldots, t_n), k-n-1 \rangle \ldots \]

.. if \( l_k = ')', and \( t_i \) \( i \leq n \) are all in \( W_f \cup V_f \)
(4) \( Q_D(L,k) = \langle L, k+1 \rangle \) if \( l_k = '1' \) or if \( l_k \in W_f \cup W_f' \).

\( T_D(L,k) = \text{true if } |L| = 1 \) and \( k = 2 \).

\( Q_D(L,k) = 1 \)

Theorem 2.2 If \( f \) (of definition 1 above) \( \in F \), then \( F_D \) (of definition III above) is also \( \in F \) and for \( d \in D \), \( F_D(\langle d \rangle, 1) = f(d) \).

Proof: The proof will be by induction on remoteness.

It will be shown that if \( l_k \) is the \( k \)-th component of \( L \) and \( l_k \in \Delta_f \cup Q_f \) then:

\( (H) \quad F_D(L,k) = F_D(L[l_k \leftarrow f(l_k)], k) \)

This is certainly true if \( l_k \) is of remoteness 0, i.e. if \( l_k \in Q_f \) because then:

\[
F_D(L,k) = F_D(L[l_k \leftarrow q(l_k)], k+1) \quad \text{by definition of } Q_D(2) \quad \text{and III 1}
\]
\[
= F_D(L[l_k \leftarrow f(l_k)], k+1) \quad \text{by definition of } f, \quad \text{III 1}
\]
\[
= F_D(L[l_k \leftarrow f(l_k)], k) \quad \text{by definition of } Q_D(4)
\]

Assume (H) is true if \( l_k \) has remoteness \( n \geq 0 \). Then if \( l_k \) is of remoteness \( n > 0 \), and \( l_k \in \Delta_f \), it follows from \( Q_D(1) \) that:

\[
F_D(L,k) = F_D(L[l_k \leftarrow '{', o_1(l_k), \ldots, o_m(l_k)(l_k), '}'], k)
\]

with \( o_i(l_k) \) for \( 1 \leq i \leq m(l_k) \) each being of remoteness \( < n \).
Rewriting the above in expanded notation and indicating \( l_k \) (= the kth component of \( L \)) by underline:

\[
F_D(L,k) = F_D(1_1, \ldots, 1_{k-1}, '1', o_1(l_k), \ldots, o_m(l_k), '1' >L', k)
\]

Then by definition \( O_D(4) \):

\[
F_D(L,k) = F_D(1_1, \ldots, 1_{k-1}, '1', o_2(l_k), \ldots, o_m(l_k), '1' >L', k+1)
\]

Since \( o_1(l_k) \in \Delta_f \cup Q_f \) and is of remoteness \( < n \) we have by \( (H) \), and then by an application of \( O_D(4) \) again:

\[
F_D(L,k) = F_D(1_1, \ldots, 1_{k-1}, '1', f(o_1(l_k)), f(o_2(l_k)), \ldots, o_m(l_k), '1' >L', k+2)
\]

And since in fact for \( i = 1 \) to \( m(l_k) \), \( o_i(l_k) \in \Delta_f \cup Q_f \), and is of remoteness \( \leq n \) by repeated application of \( (H) \) and \( O_D(4) \):

\[
F_D(L,k) = F_D(1_1, \ldots, 1_{k-1}, '1', f(o_1(l_k)), \ldots, f(o_m(l_k)), '1' >L', k+m(l_k)+1)
\]

And finally by \( O_D(3) \):

\[
F_D(L,k) = F_D(1_1, \ldots, 1_{k-1}, w(f(o_1(l_k)), \ldots, f(o_m(l_k)))) >L', k)
\]

Since \( f(l_k) = w(f(o_1(l_k)), \ldots, f(o_m(l_k))) \)

\[
F_D(L,k) = F_D(1_1, \ldots, 1_{k-1}, f(l_k) >L', k)
\]

And compacting the notation:

\[
F_D(L,k) = F_D(L[l_k + f(l_k)], k)
\]
For $L = \langle d \rangle$, $k=1$:

$$F_D(<d>,1) = F_D(<d>[d + f(d)],1)$$

$= F_D(<f(d)>,2)$ since $f(d)$ is in domain of $w_0d\langle 4 \rangle$

is applicable

$= F_D(<f(d)>,2)$ $<\langle f(d) >,2)$ is terminal so

$= f(d')$

**Properties of $f \in F$ Sufficient for Memory Efficient Implementations**

Another implementation more efficient than the two classical ones is 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:

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

$w$ = minimum, sum, 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:

1. **X** = **K**
2. For $i = 1$ to $m$
   - **Y** = $w(X,a_i)$
   - **X** = **Y**
3. 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, $a_1$. 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 \(D = \bigcup_i D_i\) is the domain of \(H\) and \(R = \bigcup_i R_i\) is its range.

The set of functions \(H\) is said to have a uniform inverse on the domain \(D\) if the functions \(H^{-1}\) and \(i_H\) defined as follows both exist on the set \(R\). If \(r\) is any member of \(R\):

\[
H^{-1}(r) = d \quad \text{where } d \text{ is the unique } e \in D \ni h_1(d) = r
\]

\[
i_H(r) = i \quad \text{where } i \text{ is the unique index } \ni h_i(d) = r \text{ for all } d \in D \text{ such that } h_i(d) = r
\]

1. 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.
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 \( D \), \( h_i(d_1) = h_j(d_2) \) 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 \( o_i \in O \) which appear in a definition \( f \in F \) 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 \( D \)
   \[
   \begin{align*}
   f(X) &= q(X) & \text{if } T(X) \\
   f(X) &= w(f(o_1(X)), \ldots, f(o_m(X)(X))) & \text{if } \overline{T}(X) \\
   \text{initially } X = d \in D
   \end{align*}
   \]

2. \( g \) on domain \( D' \)
   \[
   \begin{align*}
   g(X') &= q'(X') & \text{if } T'(X') \\
   g(X') &= w'(g(o'_1(X'), \ldots, g(o'_m(X')(X')))) & \text{if } \overline{T'}(X') \\
   \text{initially } X' = d' \in D'
   \end{align*}
   \]

If there is a 1-1 correspondence between \( D \) and \( D' \) such that whenever
d ∈ D and d' ∈ 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
\[
\begin{align*}
(1) & \quad T(\delta) = T'(\delta') \\
(2) & \quad q(\delta) = q'(\delta') \text{ if } T(\delta) \quad \text{ (and } T'(\delta')) \\
(3) & \quad m(\delta) = m'(\delta') \text{ if } \overline{T}(\delta) \quad \text{ (and } T'(\delta')) \\
(4) & \quad w = w'
\end{align*}
\]
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 \( F \):
\[
\begin{align*}
(1) \quad & \begin{cases}
  f(X) = q(X) & \text{if } T(X) \\
  f(X) = w(f(o_1(X)), \ldots, f(o_m(X)(X))) & \text{if } \overline{T}(X)
\end{cases} \\
\text{initially } X = d \in D \\
(2) \quad & \begin{cases}
  a) \quad g(X,Y) = q(X) & \text{if } T(X) \\
  b) \quad g(X,Y) = w(g(o_1(X),h_1(Y)), \ldots, g(o_m(X)(X),h_m(X)(Y))) & \text{if } \overline{T}(X) \\
  \text{initially } <X,Y> = <d,y_0> \in D' \text{ with } d \in D \text{ and } y_0 \text{ a constant}
\end{cases}
\end{align*}
\]
\( H = \{h_1, \ldots, h_m\} \) 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 \( \Gamma(x) \) 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 = \( o_i(x) \) for \( x \in \Delta_f \) and \( o_i \) is defined and \( \Gamma(x) \), 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 \( m'(X,Y) = m(X), \Gamma'(X) = \Gamma(X) \).

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 \( g(o_i(X), h_i(Y)) \) 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 \( 0, 1, 2, \ldots, n \).

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 2.3:  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 $f$. ($\rho = \langle \rho_1, \ldots, \rho_n \rangle$ is a vector which records indices, and $d$ is the initial data structure.)

$$
g(X,\rho,d) = \begin{cases} 
q(X) & \text{if } T(X) \\
g(X,\rho,d) = w(o_1(X), \langle 1 > \parallel \rho, d), \ldots, g(o_m(X), \langle m(X) \parallel \rho, d) & \text{if } T(X) \\
i initially \langle X, \rho, d \rangle = \langle d, \lambda, d \rangle \text{ with } d \in D. 
\end{cases}
$$

$g$ is strongly equivalent to $f$ by application of lemma 2.5.

Furthermore $g$ has a uniform inverse which is given by the following:
\[ i_0(X,o,d) = \rho_1 \]
\[ 0^{-1}(X,o,d) = \langle o_2 \circ \ldots \circ o_{n-1} \circ o_n(d) \rangle, \rho_1 \rho_1^{-1} \rangle, d \rangle \]

The \(0^{-1}\) 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 \(0^{-1}\) 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 \(o_i \in O\) 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 \(0^{-1}(X,o) = \langle o_i^{-1}(X), \rho_1 \rho_1^{-1}\rangle\) can be used.

This corollary can be applied to the 'Tower of Hanoi' definition ex.1.5. In that example, \(o_i \in O\) has an inverse for \(i = 1\) and \(3\) but does not quite have an inverse when \(i = 2\):

\[ o_1^{-1}(\langle x,y,z\rangle,n) = \langle x,z,y\rangle,n+1 \]
\[ o_2^{-1}(\langle x,y,z\rangle,n) = \langle x,y,z\rangle,A \quad \text{where} \quad A \text{cannot be determined from } \langle x,y,z\rangle,n \]
\[ o_3^{-1}(\langle x,y,z\rangle,n) = \langle z,y,x\rangle,n+1 \]

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.
\[ f'(\langle x, y, z, n \rangle, s) = \langle x, y \rangle \quad \text{if } n = 1 \]

\[
\begin{align*}
 f'(\langle x, y, z, n \rangle, s) &= f'(\langle x, z, y, n-1 \rangle, s) \quad \text{if } n > 1 \\
&= f'(\langle z, y, x, n-1 \rangle, s) \quad \text{if } n > 1
\end{align*}
\]

(initially \( \langle \langle x, y, z, n \rangle, s \rangle = \langle \langle 1, 2, 3, n \rangle, 0 \rangle \), \( n \in \mathbb{N} \))

Now \( f' \) is equivalent to \( f \) in 1.3 and \( o_i \) has an inverse for \( i = 1, 2, \text{ or } 3 \).

These inverses are:

\[
\begin{align*}
o_1^{-1}(\langle x, y, z, n \rangle, s) &= \langle x, z, y, n+1 \rangle, 0 \\
o_2^{-1}(\langle x, y, z, n \rangle, s) &= \langle x, y, z, s \rangle, 0 \\
o_3^{-1}(\langle x, y, z, n \rangle, s) &= \langle z, y, x, n+1 \rangle, 0
\end{align*}
\]

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

(Some unnecessary \( \rangle \)'s and \( \langle \)'s have been dropped.)

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

and the uniform inverse is given by

\[
\begin{align*}
i_0(\langle x, y, z, n, s, \rho \rangle) &= \rho_1 \\
o_1^{-1}(\langle x, y, z, n, s, \rho \rangle) &= o_1^{-1}(\langle x, y, z, n, s \rangle, o_1 \rho_1 \lambda)
\end{align*}
\]
Implementation of \( f \in F \) with Associativity and Uniform Inverse

We will first give a way of implementing any definition \( f \) in \( F \) which has a uniform inverse and is otherwise unrestricted. Then we will give a way of implementing any \( f \) in \( F \) which has a uniform inverse and in which \( w \) is associative. This is done to contrast the means necessary for implementation in these two cases. In both cases the implementation is described by a flowchart containing, as usual, interconnected assignments and decision statements. In both cases the expressions in the assignment statements and decisions are compositions involving the primitive functions and predicates \( w, q \in \Omega, m, q \) and \( T \) and the inverses \( O^{-1}, i_0 \) which enter the definition of \( f \in F \). In both cases, in addition to the above functions from the definition of \( f \), the repertoire of flow chart expression is completed by an add 1 function, a push and pop and a \( n = \) predicate. In both cases there is a storage cell \( X \) which is assumed adequate to hold any member in \( \Delta_f \cup Q_f \cup D_f \).

In the case that \( f \) has both a uniform inverse and an associative \( w \) there is also a storage list \( V \) which can hold at most any two members in \( W_f \cup w_f \). In the case that \( f \) only has a uniform inverse the list \( V \) is still necessary but it cannot be bounded in size. It may be required to hold any number of members in \( W_f \cup w_f \). The size actually used will be dependent on the specific function \( f \in F \) realized as well as the initial data-structure. In this case an auxiliary storage \( ARG \) is also used. It holds at most a number of members in \( W_f \cup w_f \) = to the largest value in the range of \( m \).

---

1. See fig. 2.1
2. See fig. 2.2
Flowcharts 1 and 2 which follow describe 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.

Flowcharts: notation and assumptions

In these flowcharts we will use the following notation. General:

(e is an expression)

\[
\begin{align*}
X & \leftarrow e & \text{the value of } e \text{ is assigned to } X \\
V & \leftarrow \text{PUSH} e & \text{the value of } e \text{ is pushed into list } V \\
X & \leftarrow \text{POP} V & \text{the top member of } V \text{ is popped and assigned to } X \\
X & \leftarrow \text{POP} V[n] & \text{the top } n \text{ members of } V \text{ are popped and assigned to } X
\end{align*}
\]

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>Flow chart</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST.KID(X)</td>
<td>0_1(X) if (X \in \Delta_f)</td>
<td></td>
</tr>
<tr>
<td>#KIDS(X)</td>
<td>m(X) if (X \in \Delta_f \cup Q_f)</td>
<td></td>
</tr>
<tr>
<td>X = TERMINAL?</td>
<td>T(X) if (X \in \Delta_f \cup Q_f)</td>
<td></td>
</tr>
<tr>
<td>PARENT(X)</td>
<td>(O^{-1}(X)) if (X \in \Delta_f); (= X) if (X \in Q_f)</td>
<td></td>
</tr>
<tr>
<td>SIB#(X)</td>
<td>i_0(X) if (X \in \Delta_f); (= 1) if (X \in Q_f)</td>
<td></td>
</tr>
<tr>
<td>NEXT.SIB(X)</td>
<td>(O_{SIB#}(X)(PARENT(X))) if (X \in \Delta_f)</td>
<td></td>
</tr>
<tr>
<td>#SIBS(X)</td>
<td>#KIDS(PARENT(X)) if (X \in \Delta_f); (= 1) if (X \in Q_f)</td>
<td></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 \(w\). This definition is used in the second flowchart following.
Flowchart 1:

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

1. $X = $ TERMINAL?
   - yes
   - no

2. $V \overset{\text{push}}{\leftarrow} q(X)$

3. $X \leftarrow $ FIRST.KID($X$)

4. $\text{SIB}\#(X) = \#\text{SIBS}(X)$?
   - yes
   - no

5. $X \leftarrow $ NEXT.SIB($X$)

6. $\text{ARG} \overset{\text{pop}}{\leftarrow} V[\#\text{SIBS}(X)]$

7. $V \overset{\text{push}}{\leftarrow} w(\text{ARG})$

8. $X \leftarrow $ PARENT($X$)

9. $X = d$?
   - no
   - yes

FIN.
Flowchart 2:

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

Figure 2.2
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 the value computed by the flowchart with traveler starting at block 0 and \( d \) in the flowchart \( = \) to \( d \) in \( f(d) \).

We now present proofs that the given flowcharts, figures 2.1 and 2.2, do in fact implement \( f \in F \) under the appropriate constraints. The proofs are very similar, both using induction on the remoteness of the data-structures in \( \Delta_f \).

Theorem 2.4: If \( f \in F \) and \( f \) has an uniform inverse then it is implemented by flowchart 1. (figure 2.1)

Proof: First we wish to show that if block 1 of flowchart (1) is entered by the traveler, with storage cell \( X \) containing data structure \( A \), and the list storage facility \( V \) containing the sequence of elements \( a \), then the traveler will eventually arrive at block 5 with the value of \( f(A) \) in \( X \), and with \( a = f(A) \) in \( V \).

This is done by induction on the remoteness of \( A \). It is obvious that if the remoteness of \( A \) is 1 then with traveler starting by entering block 1 of the flowchart with \( A \) in \( X \) and \( a \) in \( V \), the traveler will execute blocks 1, 3, 1, then 2 because \( \alpha_1(A) \) must be of remoteness 0, then 4, and if \( 1 = \text{SIB}(A) = \#\text{SIB}(A) \) then 5 will be next. After 5 the cycle 1 3 1 2 4 5 will be repeated. Altogether, the cycle repeats \( \#\text{SIB}(A)-1 \) times. Then the traveler will proceed through 1 3 1 2 4 and this time
continue with 5 7 9 to 2. Tracing the change in the content of X and V during this journey; X contains $o_{i+1}(A)$ after the $i^{th}$ cycle of 1 3 1 2 4 5; when the traveler reaches 6 X contains $o_p(A)$, but after 5 it contains A on arrival at 9. Simply tracing the blocks on travelers' path shows that V will contain $a/<f(A)>$ when traveler enters 9. Assume that if A is of remoteness $< n$, A is in X, $a$ in V, and the traveler starts by entering 1, the traveler will eventually arrive at block 9 with $f(A)$ in X and $a/<f(A)>$ in V.

Consider then that A is of remoteness $n$, A is in X, $a$ in V, and the traveler enters 1. Since X is not TERMINAL, the traveler goes to 3. As a result, the traveler enters 1 again with $o_1(A)$ in X, and $a$ still in V. Thus by the inductive hypothesis, the traveler will eventually enter 9 with $o_1(A)$ in X and $a/<f(A)>$ in V. Next since $o_1(A)$ cannot = d or the uniform inverse would not exist, the traveler will pass through 9 back to 4. Assuming that $1 = SIBf(X) = \#SIBS(X)$ where X = $o_1(A)$, then the traveler passes through 4 to 5 where X is made = to the NEXT. SIB($o_1(A)$) or $o_2(A)$. When now the traveler re-enters 1 then X is $o_2(A)$ and V is $a/<f(o_1(A))>$. Again the traveler passes through 1 and by the inductive hypotheses eventually to 9 with X containing $o_2(A)$ and V now containing $a/<f(o_1(A)), f(o_2(A))>$. This process continues until if $p = \#SIBS(A)$ traveler arrives at 1 with X containing $o_p(A)$ and V containing:
\( a \langle f(o_1(A)), f(o_2(A)), \ldots, f(o_{p-1}(A)) \rangle \)

Then by the inductive hypothesis the traveler eventually arrives at \( 9 \) with \( X \) containing \( o_p(A) \) and \( V \) containing:

\( a \langle f(o_1(A)), f(o_2(A)), \ldots, f(o_p(A)) \rangle \)

The traveler then passes through \( 3 \) entering \( 4 \), and because \( \text{SIB}(o_p(A)) = p = \text{SIB}(o_p(A)) \) the traveler will then enter \( 6 \), then \( 7 \) as a result of which \( V \) now contains:

\( a \langle f(o_1(A)), \ldots, f(o_p(A)) \rangle = a \langle f(A) \rangle \)

Then the traveler goes through \( 8 \) finally arriving at \( 3 \) with \( V \) still containing \( a \langle f(A) \rangle \)

\( \text{PARENT}(o_p(A)) = A. \)

Thus the First point to be made is proven.

Now let the traveler start by entering block \( 1 \), thus setting \( V \) to \( \lambda \) and \( X \) to \( d \in D \). The traveler then enters \( 1 \), and by the First point made above the traveler will eventually arrive at \( 9 \) with \( V = \lambda \langle f(d) \rangle = f(d) \) and \( X = d \). Then at \( 6 \) the test will succeed leaving the traveler at FINI with \( V = f(d) \).

The proof covers the case that \( d \) is of remoteness \( > 1 \).

For remoteness of \( d = 0 \), a direct trace of the flowchart will verify its adequacy.

**Theorem 2.5:** If \( f \in F \) and \( f \) has a uniform inverse and \( w \) in the definition of \( f \) is associative then \( f \) is implemented by flowchart 2.

*(figure 2.2)*
Proof: The proof is very similar to that of Theorem 2.2. The difference is in the value that will be in V when the traveler reaches \( \text{\textit{9}} \).

First we need to show that if \( A \) of remoteness \( n \), block \( 1 \) of flowchart 2 is entered with \( A \) in \( X \) and \( B \) in \( V \) then eventually the traveler arrives at \( 9 \) with \( A \) still in \( X \) and with \( V \) containing

\[
\begin{align*}
\sqrt{\ldots} & \sqrt{w[B, f(o_1(A))], f(o_2(A))}, \ldots, f(o_m(A)))] \\
& \text{by associativity} \\
& w[B, w[f(o_1(A)), \ldots, f(o_m(A))]] = w[B, f(A)]
\end{align*}
\]

Again we use induction. The case when \( A \) is of remoteness 1 is easily verifies by tracing the flowchart through the sequence of blocks \(<1 3 1 2 4 5 5 >m(A) - 1 \) times and then through \( 1 3 1 2 4 5 4 7 9 \).

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 \( 3 \) where \( X \) becomes FIRST.KID(X) = \( o_1(A) \) and the traveler returns to \( 1 \). Since \( o_1(A) \) is of remoteness \(< n \), the inductive hypothesis applies. Thus the traveler arrives at \( 9 \) with \( X \) being \( o_1(A) \) and \( V = w[B, f(o_1(X))] \). \( o_1(A) \) cannot be initial because of the inverse so the traveler goes next to \( 4 \). If we assume now that

\[
1 = \text{SIB}(X) = \#SIBS(X) \text{ where } X = o_1(A), \text{ the traveler will pass through } 4 \text{ and } 5 \text{ updating } X \text{ to contain } o_2(A) \text{ and then enter } 1 . \text{ By inductive hypothesis again the traveler will eventually arrive at } 9 \text{ with } V \text{ containing:}
\]

\[
\begin{align*}
w[B, f(o_1(A))], f(o_2(A))]
\end{align*}
\]

and \( X \) containing \( o_2(A) \). 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:

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

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 ⑤ 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 ① with these values in X and V and so by the first result the traveler will eventually arrive at ⑤ 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 flowchart 2 explicitly for this case.

The necessity for a 'uniform inverse' as opposed to a simple inverse in developing these theorems 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 (≡w(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].
3. A CLASS OF FUNCTIONS IN $F$ WITH TIME EFFICIENT IMPLEMENTATIONS

Types of Time Efficient Implementations.--Patterned Comparability

In the Introduction the existence of a time-efficient implementation of a function $f \in F$ was traced to the fact that in the standard evaluation of $f$ amongst the many sub-computations necessary, there are pairs which are virtually identical. In the implementation then it becomes possible to use the remembered result of computing one member of such a pair in computing the second member. Thus the time cost of recomputation is minimized. Examples illustrating this general assertion are given below.

Consider a function $f \in F$ for which the following properties hold.

(1) There is a relation called dominance between some pairs of members of $W_f$ (the range of $w$ in the definition of $f$) such that:

(2) Whenever two members of $W_f$ with one dominating the other both appear as arguments of a $w$ function in a given order(s), then the dominated argument may be removed--(the non-dominated one possibly requiring concurrent simple alteration) and the $w$ function will still give the same result.

Properties (1) and (2) alone are sometimes sufficient to allow significant time saving as when $w$ is a logical 'and' function with the arguments 0 or 1. If any argument of $w$ is known to be 0 then the other arguments need not be computed. However, the existence of (1) and (2) is not always sufficient to guarantee a time-saving. But if the following property also holds, time saving can be guaranteed:

(3) For a substantial set of pairs $x$ and $y$ such that $x = \sigma_f(d)$, and $y = \sigma_f(d)$ where $I$ and $J$ are sequences applicable to the same initial data structure $d$, it is simple to determine whether $f(x)$ dominates $f(y)$. (A pair of data structures $x$ and $y \in \Delta_f$ for which such a determination is possible are called comparable.)
Consider the following example of the existence of all three of these above properties. If \( w \) is a minimum, and \( W_f \) is the set of positive integers then we can define \( x \) dominates \( y \) to mean that \( x \) is less than \( y \) and (1) and (2) will be satisfied. This in itself is not enough to guarantee any time saving. Let \( \alpha \) and \( \beta \), be partial paths in an integer weighted digraph \( G \) and let 
\[
f(\alpha, c(\alpha))/f(\beta, d(\beta))
\]
be the cost of a path from node A to node B in starting with the partial path \( \alpha/\beta \) whose cost is itself \( c(\alpha)/c(\beta) \). Then with the current definition of dominance one can determine whether \( f(\alpha, c(\beta)) \) dominates \( f(\beta, c(\beta)) \) whenever \( \alpha \) and \( \beta \) end on the same node. If they do, \( f(\alpha, c(\alpha)) \) dominates \( f(\beta, c(\beta)) \) if \( c(\alpha) \geq c(\beta) \) otherwise \( f(\beta, c(\beta)) \) dominates \( f(\alpha, c(\alpha)) \). Thus (3) is satisfied and it is easy to see that dominated functions need not be computed.

In fact whenever the three properties above hold, time-savings are possible. To see this requires a brief review of implementation techniques for \( f \in F \). The standard depth first or breadth first implementations of a recursive definition \( f \in F \) is a simulation of the substitution process described in Section 2. Initially the substitution starts with the 'evaluation form' \( f(d), d \in D_f \). If \( T(d) \) is not true this is replaced by the 'evaluation form' \( w(f(c_1(d), ..., f(c_m(d)))), d) \). Then substitution is made for \( f(c_j(d)) \) for some \( 1 \leq j \leq m(d) \) is made to get a next 'evaluation form'. The process continues with subsequent substitutions for occurrences of the form \( f(a) \). Now if the above three conditions hold one can include in the evaluation an examination to determine whether two appearances of \( f \), say \( f(\alpha) \) and \( f(\beta) \) appearing in a subexpression, like \( w(f(\alpha), ..., f(\beta), ...) \).
of an evaluation form are comparable. If they are and if \( f(\alpha) \) dominates \( f(\beta) \) then \( f(\beta) \) can be eliminated from the subexpression. An entire course of substitutions is thus eliminated.

When the three conditions above hold, it is of advantage then to incorporate in the implementation a means for comparing pairs of data-structures \( \alpha \) and \( \beta \) in \( \Delta_f \). The details on how this is done will depend on the details of the definition but two broad classes can be distinguished. For a definition \( f \in F \) in which comparable data-structures arise, the pattern and frequency of their occurrences may be highly dependent on the initial data-structure, or alternately their occurrences may follow a fixed, predictable pattern largely independent of the input. In the input dependent case a facility for testing for comparability can be incorporated. It must be able to handle comparisons in a general way. Many partial results will have to be saved for comparison, even though the benefit derived from the comparison may be small. In the patterned or systematic case the implementing algorithm can often be tailored to take advantage of this fixed pattern—avoiding the need for a general comparison facility.

In this section, a significant subclass of functions in \( F \) which have a patterned structure of comparable data-structures will be studied. This subclass is called the 'explicit history' class. Corresponding to each member of this class is a set of equations whose solution is equivalent to the evaluation of the corresponding function. If this set of equations has the property of 'open-loop consistency' it can be solved by a process similar to Gaussian Elimination. This in turn will immediately provide a

* Problems whose solution can be obtained by effectively solving a set of equations with 'linear' like properties form a significant class. Such a class is carefully considered in [1]. Here we are interested in how recursive definition formulations to these and even some 'non-linear' problems are related to their set of equations formulation.
relatively efficient algorithm for implementation of such functions in $F$.

The definition of 'open-loop consistency', a property of equation sets, will be developed first then that of the explicit history function, and then in Theorem 3.1 their relation will be established.

Throughout the subsequent development we will make extensive use of notation similar to that introduced in section 2 for indicating replacements of components of a vector. Here we will extend that notation to describe replacement of a component of any expression. So if $e$ is an expression, $X_2$ a variable which may appear in $e$ and $e_1$, another expression then $e[X_2 \leftarrow e_1]$ means the result of replacing each occurrence of $X_2$ in the expression $e$ by the expression $e_1$. The notation can be further extended to cover sets of such substitutions; for example, $e[X_k \leftarrow e_k; k \in N]$ is the result of substituting the expression $e_k$ for all occurrences of $X_1$, $e_2$ for all occurrences of $X_2$, ..., and $e_n$ for all occurrences of $X_n$ in $e$. Also the notation allows composition; so $e[X_2 \leftarrow e_1][X_1 \leftarrow 0]$ is the result of first replacing each $X_2$ in $e$ by $e_1$ and then every $X_1$ in the result by 0.

Note that the expression $e[X_2 \leftarrow e_1][X_1 \leftarrow 0] [X_1 \leftarrow 0]$ as well as the expression $e[X_1 \leftarrow 0][X_2 \leftarrow e_1[X_1 \leftarrow 0]]$ gives the same result as that of the notation in the previous sentence. Such reorderings yielding the same result will be used in the proof of Theorem 3.1.

Open-Loop Consistency and Explicit History Definitions

Equation Sets

$\mathcal{W}$ is a set of functions. If $w \in \mathcal{W}$, then $w$ is defined on $0$ arguments, $1$ argument, ..., $n$ arguments. Each argument is drawn from a set $S$. 


The range of each $w$ is also $S$. $S$ contains a 0 element with the property that $w(X_1, \ldots, X_{c-1}, X_c, X_{c+1}, \ldots, X_n) = w(X_1, \ldots, X_{c-1}, X_c, X_{c+1}, \ldots, X_n)$ if $X_c = 0$.

$E_n(W_n)$ is a set of equations involving $n$ functions from $W_n = \{w_1, \ldots, w_n\}$, and the variables $X_1, \ldots, X_n$.

$E_n(W_n)$ contains a subset called the **basis** subset.

The basis of $E_n(W_n) = \{X_j = w_j(X_{j_1}, \ldots, X_{j_m}) | j \in N, j_1 \in N, j_k > j_i \text{ if } i > k\}$.

In addition, $E_n(W_n)$ satisfies the closure conditions that:

**C1:** If $X_j = e$ is in $E_n(W_n)$ then so is $X_j = e[X_k = 0]$ for any $X_k$ in $e$.

and:

**C2:** If for any given $j$ and $k \in N, X_j = e_1$ and $X_k = e_2$ are each members of $E_n$, then so is $X_j = e_1[X_k \neq e_2]$.

$E_n(W_n)$ consists only of those equations in the basis together with those constructable from C1 and C2.

From here on we use $E_n$ to stand for $E_n(W_n)$.

**Open-Loop Consistent Equation Sets**

Let $Q = \{X_j = e_j | j \in N\}$ be a set of equations in $E_n$.

Let $Q_c = \{X_j = e_j | 1 \leq j \leq c-1\} U \{X_c = e_c[X_c \neq 0]\} U \{X_j = e_j | c+1 \leq j \leq n\}$.

If every solution to $Q$ is also a solution to $Q_c$, then $Q$ in $E_n$ is open-loop consistent in $X_c$. If $Q$ of $E_n$ is open-loop consistent in all variables $X_c, c \in N$ then $Q$ is open-loop consistent in $E_n$. If every subset in $E_n$ of the form of $Q$ is open-loop consistent, then $E_n$ is open-loop consistent.
The significance of open-loop consistency follows from the following.

On the one hand, if the basis set of $E_n$ is:

$$\{ X_i = e_i | i \in N \}$$

and $E_n$ is open-loop consistent then a process analogous to Gaussian Elimination will be adequate to solve the basis equations for $\{ X_i | i \in N \}$. This process is based on the alternate application of two operations applied to a set of equations which is initially the basis of $E_n$. The first operation is used to remove the recursive appearance in an equation of the form $X_j = e_j$ of any occurrence of $X_j$ in $e_j$. This is made possible if the set of equations is Open-Loop Consistent, by setting all occurrences of $X_j$ in $e_j$ to 0. (Open-loop consistency is a specialization of a considerably more general property which would allow removal of such recursive appearances, which we hope to develop at a future time.) This first operation is only necessary if there is such a recursive appearance to be removed in one or more equations of the set. The second operation uses an equation of the form $X_j = e'_j$ with $X_j$ appearances removed from $e'_j$ to substitute for all appearances of $X_j$ on the right of other equations in the set, thus eliminating all occurrences of $X_j$ on the right of all equations in the set. The two operations can be repeated $n$ times as needed to produce a solution to the basis set from $E_n$. Note that every equation in the sets that result from different steps in this process is in $E_n$ so that the open-loop consistency condition is always applicable. In any case this process provides a relatively efficient means of solving such a set of equations.

On the other hand the evaluation of an 'explicit history' recursive definition will be shown to be equivalent to the solution of a corresponding
set of equations forming the basis of a set $E_n$ if $E_n$ is open-loop consistent. Thus the relatively efficient solution will be applicable to 'explicit history' definitions having this property.

Explicit History Recursive Definitions

The 'explicit history' recursive definition will initially be given in a form which allows the simple description of the corresponding set of equations. Later we will show other equivalent forms for such definitions.

Recalling that $N = \{1, 2, \ldots, n\}$, let $H$ be a vector all of whose components are in $N$, and no two of which are the same. $H$ is the history vector. $H/c$ will be used as a shorthand for $H/<c> = \text{the concatenation of}<c>$ with vector $H$. If $V$ is a vector, $\{V\}$ is the set of components of $V$, $\mathcal{W}_i \subset \mathcal{W}_n$, a set of functions defined earlier. $c_i \in N$, $c_j > c_i$ if $i > j$.

An 'explicit history' recursive definition has the form:

\[
\begin{cases}
  x^H_c = 0 & \text{if } c \in \{H\} \\
  x^H_c = w_c(x^H_{c_1}, \ldots, x^H_{c_m}) & \text{if } c \in N - \{H\} \\
  \text{initially } H = \lambda, c \in N
\end{cases}
\]

If $f$ is an explicit history recursive definition then the corresponding set of equations designated $E_n(f)$ is built on the following basis:

\[
\text{def } 5.1.1 \{ x_c = w_c(x_{c_1}, \ldots, x_{c_m}) = c_c | c \in N \}
Relation of Open-loop Consistent Equation Sets and Explicit History Definitions

Theorem 3.1:  If \( f \) is an explicit history recursive definition and \( E_n(f) \) its corresponding equation set is open-loop consistent then the set of values \( \{ X^\lambda_c \mid c \in N \} \) as determined by evaluating \( f \), will satisfy the basis of \( E_n(f) \) with \( X^\lambda_c = X^\lambda_c \) for all \( c \in N \).

Proof: A partly inductive argument will be used.

First we define two kinds of expressions \( Z_j^\alpha \) and \( Y_j^\alpha \):

1) \( Z_j^\alpha = e_j = \text{the right side of the equation } X_j = e_j \text{ in } E_n \)
   
2) \( Y_j^\alpha = 0 \)

If \( j \in N - \{ \alpha \} \):

3) \( Z_j^\alpha = \text{an expression giving } X_j \text{ as a composition of functions} \)
   
4) \( Y_j^\alpha = Z_j^\alpha \left[ X_j = 0; k \in \{ \alpha \} \right] \)

\( Z_j^\lambda \) then gives \( X_j \) as an expression involving no variables and thus as a composition of functions from \( W_n \) with no arguments. \( w_j() \) must be defined as a constant and so the expression \( Z_j^\lambda \) must be evaluated as a constant. By definition of \( Y_j^\alpha \):

\( Y_j^\lambda = Z_j^\lambda \)

Note that by definition when \( N - \{ \alpha \} = \{ j \} \) the equation \( X_j = Z_j^\alpha \) is in \( E_n \); thus since \( E_n \) is open-loop consistent so is \( X_j = Y_j^\alpha \) (by \( d_2 \) above).

(H) Assume that for \( \alpha = \text{any sequence of } k < n \text{ integers taken from } N \) with no two equal and \( c \in N - \{ \alpha \} \) that:

\( X_j = Z_j^{\alpha/\{c\}} \quad ; j \in N - \{ \alpha/\{c\} \} \)

are each true equations in \( E_n \).
Then for $c \in N$:

$$X_c = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0]$$ is in $E_n$ by def 3.1.1

$$X_c = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0]$$ is in $E_n$ by definition of $E_n$ and $H$

$X_c = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0]$ is in $E_n$ because $E_n$ is open-loop consistent and $X_j = Z^a_{j\in\alpha\not\in\langle a\rangle}$ is in $E_n$ by $H$.

$X_c = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0]$ is in $E_n$ by reordering and substitution.

$X_c = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0]$ is in $E_n$ by definition $d_2$, by which $Z^a_{j\in\alpha\not\in\langle a\rangle} = 0$ when $j \in \{a\not\in\langle a\rangle\}$.

So if $Z^a_{j\in\alpha\not\in\langle a\rangle}$ is defined as:

$$Z^a_{c} = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0]$$

then $X_c = Z^a_{c}$ is true and is in $E_n$ for $c \in N-a$ and by definition of $Y^a_{c}$ and $d_2$:

$$Y^a_{c} = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0:k \in \alpha]$$

by $d_2$

$$Y^a_{c} = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0:k \in \alpha]$$

by reordering

$$Y^a_{c} = e_c[X_j + Z^a_{j\in\alpha\not\in\langle a\rangle} + X_c = 0:k \in \alpha]$$

by $d_2$

or since by $d_1$:

$$Y^a_{c} = e_c[X_j + Y^a_{j\in\alpha\not\in\langle a\rangle}]$$

These two relations give the recursive definition we seek.

The recursive definition, def 3.1, is not, as written, in the form required for a member of $F$. However, by simple transformations a definition
which is a member of F can be generated.

Let \( f(H,c) = \chi^H_c \) and make the subscript of \( w_c \) an argument of a modified \( w \) function.

\[
\begin{align*}
    f(H,c) &= 0 & \text{if } c \in \{H\} \\
    f(H,c) &= w(c,f(H/H,c_1), \ldots, f(H/H,c_m)) & \text{if } c \in N\{H\} \\
    \text{initially } H = \lambda, c \in N
\end{align*}
\]

Still one more small change is needed to create a definition in F. The first argument \( c \) of \( w \) is not in the required form. It is put in the correct form by adding another argument, \( s \), which is either 0 or 1.

\[
\begin{align*}
    f(H,c,0) &= 0 & \text{if } c \in H \\
    f(H,c,s) &= c & \text{if } s = 1 \\
    \text{def 3.1b} \quad f(H,c,s) &= w(f(H,c,1),f(H/H,c_1,0), \ldots, f(H/H,c_n,0)) & \text{if } c \in N\{H\} \\
    \text{initially } H = \lambda, c \in N, s = 0
\end{align*}
\]

Theorem 5.1 is the Main Result of this section. Its application will be illustrated by an example. In order, however, to make this result generally applicable, it would be necessary to further elucidate two questions concerning the set of equations which correspond to a given 'explicit history' definition.
The first of these is: do sets of equations which are open-loop consistent arise with significant frequency in practice. The second question arises after one has an open-loop consistent set at which point one would like to know if the solution is unique. This would guarantee that evaluation of the recursive definition would give the same result as would any procedure for solving the corresponding equation set. If the solution is not unique it would be well to know to which solutions of the equation set, obtained by what procedure, the evaluation of the corresponding recursive definition (which is always unique) is equivalent. These two questions are not considered further here.

Now, however, we note some very special cases in which the answers to these questions are evident.

We say $E_n$ has no loop if the equations in the basis of $E_n$:

$$x_i = \{e_i \mid i \in N\}$$

have the property that $e_i$ contains no occurrence of any variable $x_{i-k}$, $k \geq 0$. If this is the case then it is easy to see that no equation in $E_n$ can be recursive - i.e. have the variable appearing on the left of the equation also occur on the right. From this it follows directly that $E_n$ is open-loop consistent. Furthermore, it is easy to show that in this case the solutions are unique.

**Lemma 3.1**: If the basis of $E_n$ has no loops then $E_n$ is open-loop consistent and its solution is unique.

**Examples of Applications of Theorem 3.1**

The maximum and minimum path problems, each defined on a directed graph will be used to illustrate the application of Theorem 3.1.
The following graph definition will be used:

$G$ is a directed graph with a set of nodes $N = \{1, 2, \ldots, n\}$

If $c$ is a node in $G$ then:

- $c_i$ is the $i^{th}$ neighbor of $c$ (the $i^{th}$ node reachable from $c$ by
  traversing a single directed branch
- $c_m$ is the last neighbor of $c$

$W(c_i)$ is the weight of the branch from $c$ to $c_i$. It is always $> 0$.

If $X_i$ is a number then:

- $\max(X_1, \ldots, X_n) = $ the maximum of $X_i$, $i \in N$
- $\min(X_1, \ldots, X_n) = $ the minimum of $X_i$, $i \in N$

A path $h = <h_1, \ldots, h_p>$ in $G$ is a sequence of nodes in $G$ such

that there is a branch $<h_i, h_{i+1}>$ in $G$ for each $p-1 \leq i \leq 1$. The

weight of path $i$ is the sum of the weights of the branches

$<h_i, h_{i+1}, \ldots> \geq i \geq 1$.

Now if we interpret $\text{MAW}(h, c)$ to be the maximum weight loopless path
starting with the path $h/\langle c \rangle$ and ending on node $t$, since such a path must
next pass to some neighbor, $c_i$, of $c$ which to avoid loops is not already
$h/\langle c \rangle$ we have that:

\[
\text{Ex. 3.1} \quad \begin{cases} 
\text{MAW}(h, c) = 0 & \text{if } c \in \{h\} \\
\text{MAW}(h, c) = \max(W(c_i) + \text{MAW}(h/c, c_i), \ldots, W(c_m) + \text{MAW}(h/c, c_m)) & \text{if } c \in N - \{h\}
\end{cases}
\]

Initially $h = \lambda, c \in N$

and analogously for the minimum path we have if $\text{MIW}(h, c)$ is the minimum
weight of all paths starting with path $k$ and ending on node $t$, then its
recursive definition can be given by the equations:
Ex 3.2

\[
\begin{aligned}
\text{MIW}(h,c) &= 0 & \text{if } c \in \{h\} \\
\text{MIW}(h,c) &= \min(W(c_1)+\text{MIW}(h/c,c_1), \ldots, W(c_m)+\text{MIW}(h/c,c_m)) & \text{if } c \in N-\{h\} \\
\text{initially } h &= \lambda, c \in N
\end{aligned}
\]

With \(w(c,X_1, \ldots, X_m) = \max/\min(W(c_1)+X_1, \ldots, W(c_m)+X_m)\), both examples are in the form of definition 3.1a and thus are equivalent to the recursive definition in theorem 3.1. Therefore, in both cases we may speak of the corresponding set of equations def 3.1.1. Consider the equations in the basis of \(E_n(\text{MAW})\) corresponding to ex 3.1 first:

Ex 3.1.1 \( \{x_c = \max(W(c_1)+X_{c_1}, W(c_2)+X_{c_2}, \ldots, W(c_m)+X_{c_m}) | c \in N \} \)

where as noted \(c_i\) is the \(i\)th neighbor of node \(c\) in \(G\). Now in general because of the \(\max\) function, \(E_n(\text{MAW})\) with the above basis will not be open-loop consistent. In fact if the basis of \(E_n(\text{MAW})\) contains an equation in which the same variable appears on the right then \(E_n(\text{MAW})\) is not open-loop consistent because suppose \(c_j = c\), \(a_{c_i} > 0\) and:

(a) \(x_c = \max(a_{c_1}+X_{c_1}, \ldots, a_{c_j}+X_{c_j}, \ldots, a_{c_m}+X_{c_m})\)

then if \(E_n(\text{MAW})\) was open-loop consistent \(x_c\) would also satisfy:

(b) \(x_c = \max(a_{c_1}+X_{c_1}, \ldots, a_{c_{j-1}}+X_{c_{j-1}}, a_{c_j}+X_{c_j}, \ldots, a_{c_m}+X_{c_m})\)

but it is easy to see by substitution for \(x_{c_j} = x_c\) using (b) in (a) that this cannot be.

However, if the graph \(G\) does not have any loops, then \(E_n(\text{MAW})\) is open-loop consistent, since then the conditions of lemma 3.1 are met. So if \(G\) is loopless, theorem 3.1 can be applied and solutions to the equations of ex 3.1.1 will be equivalent to an evaluation of the recursive definition in ex 3.1. The number of equations in ex 3.1.1 is equal to the number of nodes in th.
graph $G$. No variable appears on both the left and right sides of an equation and cannot do so as the result of any substitutions because $G$ has no loops. One of the equations in the set, say $X_n = e_n$, will have only a constant on its right side, i.e. $e_n = a_n$. This constant may be substituted for all occurrences of $X_n$ on the right of the other equations. As a result, a second equation, say $X_{n-1} = e_{n-1} [X_n + a_n] = a_{n-1}$ will have only a constant on its right. This process continues until all resultant equations up to the one with the variable whose value is sought have only a constant on their right. The $i^{th}$ substitution involves substituting for at most $i$ variables, $i$ additions and taking the maximum over $i$ numbers, or the order of $i$ steps. $i$ ranges from 1 to the number of equations at most. So after the equations are set up, the solution algorithm is has a complexity of the order of $n^2$.

In eq 3.2, the set of equations $E_n$ (MIW) is open-loop consistent.

This is easily seen for if $c_j = c$, $a_j > 0$ and:

(a) $X_c = \min(a_{c_1} + X_{c_1}, \ldots, a_{c_j} + X_{c_j}, \ldots, a_{c_m} + X_{c_m})$

and consider:

(b) $X_c = \min(a_{c_1} + X_{c_1}, \ldots, a_{c_j-1} + X_{c_{j-1}}, a_{c_j} + X_{c_j}, \ldots, a_{c_m} + X_{c_m})$

substituting with (b) in the right of (a) we get:

$X_c = \min(a_{c_1} + X_{c_1}, \ldots, \min(a_{c_1} + X_{c_1}, \ldots, a_{c_{j-1}} X_{c_{j-1}}, a_{c_j} X_{c_j}, \ldots, a_{c_m} X_{c_m}), \ldots, a_{c_m} + X_{c_m})$

which because of the properties of $\min$:

$X_c = \min(a_{c_1} + X_{c_1}, \ldots, a_{c_{j-1}} X_{c_{j-1}}, a_{c_{j+1}} X_{c_{j+1}}, \ldots, a_{c_m} + X_{c_m})$

The same equation would result from substituting for $X_c$ on the left of (a) with the right of (b). Thus (b) satisfies (a) verifying the claimed
open-loop consistency. Not only is \( E_n^{(MIW)} \) open-loop consistent, but the basis of \( E_n^{(MIW)} \) can be shown to have a unique solution.

Thus a substitutional solution to this set of equations will give a solution to the corresponding recursive definition. This solution has a time complexity \( \leq n^3 \), in part because after each substitution step the terms can be gathered in this case so they will again be a simple-min of constants + 'ed with variables, in the same form as the initial basis equations. Furthermore in this case, again - primarily because of the properties of the min function, the complexity of the gaussian elimination solution can be reduced to order \( n^2 \) by adopting the appropriate order of substitution. The appropriate order is given by the following considerations.

First the \( X_t \) equation will be:

\[
X_t = 0
\]

The right side is a constant. So substitute throughout the other equations right sides this constant value for \( X_t \). In these other \( n-1 \) equations carry out the + and min operations with the substituted value. That will leave the right side of these equations again in the form of a min of terms. All except one of these terms being a constant + 'ed with a variable. The exceptional term being simply a constant. The variable \( X_t \) will not appear on the right side of any equation. The key step now is to examine the \( n-1 \) equations for the one whose constant term is the smallest.

Say this is:

\[
X_j = \min(c_{j_1} + X_{j_1}, \ldots, c_{j_n} + X_{j_n}, d_j)
\]

where \( c_{j_1} \) and \( d_j \) are constants. Then that equation can be simply replaced by:

\[
X_j = d_j
\]
That is, all the variables can be removed from the right side of this equation. Roughly this is justified because, since $d_j$ is the smallest constant in the set of equations and consideration of the substitutional method of solution indicates that no variable can ultimately be assigned a lower value. Next in the $n-2$ equations (excepting the $X_i$ and $X_j$ equations which are solved) $d_j$ is substituted for all appearances of $X_j$. Terms are then gathered in the remaining $n-2$ equations. Again the one of those having the smallest constant is chosen and all its variable terms removed, and so forth thus providing the solution for another variable. The process is repeated until the variable whose value is sought becomes the one having the smallest constants on the right, of all remaining unsolved equations. That is the value of the variable.

In this method of solution one is always substituting a constant thus greatly reducing the effort necessary to simplify the right sides of equations. It is essentially Dykstra's algorithm and its complexity of order $n^2$.

Some Simple Equivalences

Throughout this paper we have presented recursive definitions purported to be 'natural'. In fact, however, for any given algorithm design problem one may develop a number of recursive definitions each being equally entitled to being called natural. The differences in these equally natural descriptions may range from very small details to deeper differences representing radically different points of view. We have seen some small adjustments of detail in getting equivalent formulations of the recursive definition of theorem 3.1. Deeper differences are represented by the
alternate formulations for the recursive definition of the set of binary numbers given in the introduction. The path examples of the previous pages were designed to look natural - but also to fit the theorem formulation - many other formulations - which did not quite fit the theorem formulation could have been represented as natural. There is no deception here; they are all natural. There are many equivalent ways of defining the same function. A number of simple equivalences will now be developed which account for some common alternatives. Possession of such equivalences allow one to move easily from definition to definition. As we have seen, some definitions are closer to good algorithms than others. Some definitions, unary ones, can even be considered as specifying algorithms. In fact, all the theorems given thus far can be viewed as equivalences which allow one to move from an initial definition to a better one. The equivalences given now differ only in being somewhat simpler than those developed previously, and seeming to have more to do with initial formulation.

The equivalences to be given all involve trade-offs between the complexity of the \( w \) and \( o \)-functions, or between the complexity within the argument of the function being defined (say \( f \)) and the complexity outside that argument. If the structure of the argument of \( f \) is considered the data-structure, and the way in which \( f \) enters as an argument (of \( w \)) in the recursive definition is called the control structure, then these equivalences give trade-offs between data and control structures.

The significance of the following two theorems can probably be better appreciated if the example following the theorems are scanned before the theorems and proofs are read.
Theorem 5.2: If operations $\cdot$ and $+$ have the properties:

1. $X_1 \cdot (X_2 \cdot X_3) = (X_1 \cdot X_2) \cdot X_3$

2. $X \cdot (\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} (X \cdot X_i)$

and if:

- $g(X) = q(X)$ if $T(X)$
- $g(X) = \sum_{i=1}^{m} (t_i(X) \cdot g(o_i(X)))$ if $T(X)$

Initially $X \in D$

and:

- $f(y, X) = y \cdot q(X)$ if $T(X)$
- $f(y, X) = \sum_{i=1}^{m} f(y \cdot t_i(X), o_i(X))$ if $T(X)$

Initially $X \in D$

then for each $X \in \Delta_g$:

$$f(y, X) = y \cdot g(X)$$

Proof: We simply show that $y \cdot g(X)$ with $g(X)$ as defined in $d_1$ satisfies
or makes true the relations of $d_2$ when appropriately substituted
for $f(y, X)$ in $d_2$.
Thus if $X$ is of remoteness 0, $d_2$-1 defines $f(y, X)$ so that sub-
stituting:

$$y \cdot g(X) = y \cdot q(X)$$

$$y \cdot q(X) = y \cdot q(X)$$ since $g(X) = q(X)$ when $X$ is of remote-
ness 0 by $d_1$

For $X$ having remoteness > 0, $d_2$-2 defines $f(y \cdot X)$. Substitute
$y \cdot g(X)$ for $f(y \cdot X)$ throughout that equation to determine if it
is thus satisfied. Note that if $X \in \Delta_g$ then certainly $o_i(X)$
is also a member of $\Delta_g$. 

\[ y \cdot g(X) = \sum_{i=1}^{m(X)} [(y \cdot t_i(X)) \cdot g(o_i(X))] \]
\[ = \sum_{i=1}^{m(X)} [y \cdot (t_i(X) \cdot g(o_i(X)))] \quad \text{by p1} \]
\[ = y \cdot \sum_{i=1}^{m(X)} [t_i(X) \cdot g(o_i(X))] \quad \text{by p2} \]
\[ y \cdot g(X) = y \cdot g(X) \quad \text{by d1} \]

Our first example of the application of theorem 3.2 requires some definitions. Let \( B = \{b_1, \ldots, b_m\} \) and \( A = \{a_1, \ldots, a_n\} \), both be sets whose components are vectors:
\[ B \uplus A = \{b_1/a_1, \ldots, b_m/a_n\} \cup \{b_2, \ldots, b_m\} \uplus A \]
With this definition of \( \uplus \), the following property clearly holds for \( A, B \) and \( C \) each sets of vectors:
\[ p1 \]
\[ C \uplus (B \uplus A) = (C \uplus B) \uplus A. \]
Also if \( \cup \) is the usual union operation then if each \( X_i, i \in N \), as well as \( A \) is a set of vectors:
\[ p2 \]
\[ X \uplus (X_1 \cup X_2 \cup \ldots \cup X_m) = (X \uplus X_1) \cup (X \uplus X_2) \cup \ldots \cup (X \uplus X_m) \]
Now the set of all \( n \) bit binary numbers (vectors) = \( B(n) \) is the set of all \( n-1 \) bit binary numbers, \( B(n-1) \), with a 0 attached in front of each member of the set, \( \{<0>\} \uplus B(n-1) \), together with \( \{<1>\} \uplus B(n-1) \) with a 1 attached in front of each member of the set, \( \{<1>\} \uplus B(n-1) \).

More formally the definition is:
\[
\begin{cases}
B(n) = \{\lambda\} & \text{if } n = 0 \\
B(n) = (\{<0>\} \uplus B(n-1)) \cup (\{<1>\} \uplus B(n-1)) & \text{if } n > 0
\end{cases}
\]
initially \( n = \) the number of bits of the binary number

Since \( \uplus \) and \( \cup \) satisfy \( p1 \) and \( p2 \), it follows by theorem 3.2 that
B'(y,n) given in the following definition is a member of \( \mathcal{F} \) and is related to the definition above in that \( B'(y,n) = y \{ \lambda \} B(n) \).

\[
\begin{align*}
B'(y,n) &= y \{ \lambda \} = y & \text{if } n = 0 \\
B'(y,n) &= B'(y \{ \{0\} \}, n-1) \cup B'(y \{ \{1\} \}, n-1) & \text{if } n > 0 \\
\text{initially } n &= \text{the number of bits of the binary number, } y = \{ \lambda \}
\end{align*}
\]

(2)

Note that (2) is essentially the definition given in the Introduction for binary numbers (ex 1.1).

As a second example consider the application of this theorem to the previous path examples. Thus equivalent to the definition of ex 3.1 we have by theorem 3.2 since \( \max(\text{corresponding to } +) \) and \( +(\text{corresponding to } \cdot) \) have the appropriate properties the following function \( \text{MAW} \) (the pair \( \langle h, c \rangle \) corresponds to \( X \), \( W(c_i) \) corresponds to \( t_i(X) \), \( \langle h, c \rangle, c_1 \rangle \) corresponds to \( c_i(X) \)).

\[
\begin{align*}
\text{MAW}'(y, h, c) &= y + 0 & \text{if } c \in \{ h \} \\
\text{MAW}'(y, h, c) &= \max(f(y + W(c_i), h / c, c_1), \ldots, f(y + W(c_m), h / c, c_m)) & \text{if } c \in N \setminus \{ h \} \\
\text{initially } h &= \lambda, c \in N, y = 0
\end{align*}
\]

The next equivalence allows the replacement of a vector in the data-structure by an individual component at the expense of a more complex control structure.

Definitions:

* \( \alpha \) is a sequence of symbols in the alphabet \( A \).

If \( a \in A \), \( R_i(a) \) for \( i \leq m(a) \) is a sequence of symbols in \( \alpha \). There is a subset of \( A \), say \( A' \), such that for each \( a' \in A' \), \( m(a') = 1 \), and \( R_i(a') = \lambda \).

This is necessary if the definition of \( f \) in the following theorem is to be legitimate.
Theorem 3.3: Given the binary operations $\cdot$ and $+$ with the following properties:

\[ p_1: \ (X_1 \cdot X_2) \cdot X_3 = X_1 \cdot (X_2 \cdot X_3) \]
\[ p_2: \ (L_i X_i) \cdot Z = L_i (X_i \cdot Z) \]
\[ p_3: \ X \cdot X \cdot 0 = X \]

and given the definition:

\[
d_1 \begin{cases} 
  f(a) = 0 & \text{if } a = \lambda \\
  f[a] = \sum_{i=1}^{n} r_i(a) \cdot f[a \cdot R_i(a)] & \text{if } a \neq \lambda \\
  \text{initially } a = <a> \text{ where } a \in A
\end{cases}
\]

then:

**fact1:** $f(a) = f(a_{i:j}) \cdot f(a_{j+1:n})^{*}$ \hspace{1cm} if $j \in \mathbb{N}$

and further given the definition:

\[
d_2 \begin{cases} 
  g(a) = 0 & \text{if } a \in A' \\
  g(a) = \sum_{i=1}^{n} r_i(a) \cdot g([R_i(a)]_1) \cdot g([R_i(a)]_2) \cdot \ldots \cdot g([R_i(a)]_{n_i}) & \text{if } a \notin A'
\end{cases}
\]

initially $a \in A$

then:

**fact2:** for each $a \in A$, $f(<a>) = g(a)$

Proof: First we show that:

\[ f(a) = f(a_{i:j}) \cdot f(a_{j+1:n}) \]

(H) Assume since $a[a_{i} + R_i(a)]$ is of lesser remoteness than $a$ that

**fact1** is true, i.e.:

\[ f(a[a_{i} + R_i(a)]) = f(R_i(a)) \cdot f(a_{2:n}) \]

Thus:

\[ f(a) = \sum_{i=1}^{n} (r_i(a) \cdot (f(R_i(a)) \cdot f(a_{2:n}))) \]
\[ = \sum_{i=1}^{n} ((r_i(a) \cdot f(R_i(a))) \cdot f(a_{2:n})) \hspace{1cm} \text{by } p1 \]
\[ = (\sum_{i=1}^{n} r_i(a) \cdot f(R_i(a))) \cdot f(a_{2:n}) \hspace{1cm} \text{by } p2 \]
\[ - f(a) \cdot f(a_{2:n}) \]

\[ * \quad a_{i:j} = <a_1, a_{i+1}, \ldots, a_j> \text{ if } a = <a_1, \ldots, a_n>; \quad a_{i:i} = <a_i> \]
The basis for this induction is given next.

If \( a \) is of remoteness 1 then: \( a[a_1 + R_i(a)] \) must be of remoteness 0, must = so:

\[
\begin{align*}
\hat{f}(a) &= \sum_{i=1}^{m(a)} r_i(a_1) \cdot f(a[a_1 + R_i(a)]) \\
&= \sum_{i=1}^{m(a)} r_i(a_1) \cdot f(\lambda) \\
&= \sum_{i=1}^{m(a_1)} r_i(a_1) \quad \text{since } f(\lambda) = 0 \text{ by } d_1 \text{ and } p5
\end{align*}
\]

Since \( a[a_1 + R_i(a)] = \lambda \quad a \) must = \( <a'> \) with \( a' \in A' \)

Therefore if \( a \) is of remoteness 1 then \( a = <a'>, a_1 = a', \)

\( a_{2:n} = \lambda, \) and:

\[
\begin{align*}
\hat{f}(a) &= \hat{f}(<a'>) = \sum_{i=1}^{m(a')} r_i(a') \\
&= \sum_{i=1}^{m(a')} r_i(a') \cdot f(\lambda) \quad \text{by } p1 \\
&= \hat{f}(<a'>) \cdot f(a_{2:n})
\end{align*}
\]

(H1) Now assume that for \( 1 \leq j \leq m \) that:

\[
\hat{f}(a) = \hat{f}(a_{j}; j) \cdot \hat{f}(a_{j+1}; n')
\]

this has already been shown true when \( j = 1 \)

then:

\[
\hat{f}(a_{j+1}; n) = \hat{f}(a_{j+1}) \cdot \hat{f}(a_{j+2}; n')
\]

so:

\[
\begin{align*}
\hat{f}(a) &= \hat{f}(a_{1}; j) \cdot (\hat{f}(a_{j+1}) \cdot \hat{f}(a_{j+2}; n')) \quad \text{by assumption (H1)} \\
&= [\hat{f}(a_{1}; j) \cdot \hat{f}(a_{j+1})] \cdot \hat{f}(a_{j+2}; n') \quad \text{by } p2 \\
&= \hat{f}(a_{1}; j+1) \cdot \hat{f}(a_{j+2}; n) \quad \text{by (H1)}
\end{align*}
\]

This completes the proof of fact1, fact1 may be used to establish

fact2. By \( d_1 \) if \( a \not\in A' \):

\[
\begin{align*}
\hat{f}(<a>) &= \sum_{i=1}^{m(a)} r_i(a_1) \cdot f(R_i(a)) \\
&= \sum_{i=1}^{m(a_1)} r_i(a_1) \cdot (f([R_i(a)]_1) \cdot \ldots \cdot f([R_i(a)]_n))
\end{align*}
\]
and if \( a \in A' \):

\[ f(a) = 0 \]

Note that these last two equations for \( f(a) \) are almost identical to the equations for \( g(a) \) in \( d_2 \). A simple inductive argument based on this near identity shows that:

\[ f(a) = g(a) \]

---

Example of Application of Theorem 3.2

This example involves a problem which together with its 'good' solution is discussed in \([3]\). We will see how one could get from an initial formulation of a recursive solution to this problem to the 'good' algorithm of \([3]\) via theorem 3.2. The problem involves the multiplication of a set of matrices; \( M_1 \times M_2 \times \ldots \times M_n \). The dimensions of each matrix is given. The dimensions of \( M_i \) are \((r_i, c_i)\). The number of multiplications necessary to multiply \( M_i \) by \( M_j \) is then \( r_i \times (c_i = r_j) \times c_j \). The multiplication \( M_1 \times M_2 \times \ldots \times M_n \) may be associated in any way to get the same answer; so \((M_1 \times M_2) \times M_3 = M_1 \times (M_2 \times M_3)\) for example, but with differing numbers of multiplications required. The problem is to design an algorithm to find an association which will give the minimum number of multiplications. In fact, we will design an algorithm to find that minimum number of multiplications - which is the heart of the matter. The initial approach is to somehow enumerate all ways of associating \( M_1 \times \ldots \times M_n \) and the corresponding costs in number of multiplications and then to choose the association giving the minimum of these. Different associations are equivalent to different ways in which the expression \( M_1 \times \ldots \times M_n \) can be parenthesized. The last matrix multiplication in any such association must involve one of the following alternatives:
$M_1$ with the result of $(M_2 \times \ldots \times M_n)$ with a cost of

$\tau_1 \times (c_1 = \tau_2) \times c_n + \text{the cost (}=0) \text{ of doing } (M_1 \times M_1) + \text{the cost of doing } (M_2 \times \ldots \times M_n) \text{ or }$

The result of $(M_1 \times M_2)$ with the result of $(M_3 \times \ldots \times M_n)$

with a cost of $\tau_1 \times (c_2 = \tau_3) \times c_n + \text{the cost of doing } (M_1 \times M_2) + \text{the cost of doing } (M_3 \times \ldots \times M_n)$ or \ldots

The result of $(M_1 \times \ldots \times M_{n-1})$ with $M_n$ with a cost of

$\tau_1 \times (c_{n-1} = \tau_n) \times c_n + \text{the cost of doing } (M_1 \times \ldots \times M_{n-1}) + \text{the cost (}=0) \text{ of doing } (M_n \times M_n)$.

If $a = (i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)$ and $f(a)$ = the cost of doing

$(M_{i_1} \times \ldots \times M_{j_1}) + \text{the cost of doing } (M_{i_2} \times \ldots \times M_{j_2}) + \ldots$ + the cost of doing $(M_{i_n} \times \ldots \times M_{j_n})$ then:

\[
\begin{align*}
\text{if } a &= \lambda \\
f(a) &= f(a[\alpha_1 + \lambda^*]) \\
f(a) &= \text{Min}(\{f(\alpha + \langle i_1, i_1 \rangle), \ldots, f(\alpha + \langle i_1, i_1 \rangle) \}) \\
\text{initially } a &= (\langle 1, n \rangle)
\end{align*}
\]

Let $a = (i, j)$ in which $i, j \in N$ and $j \geq i$. If $m(a) = 1$ and $R_k(a) = \lambda$

when $i = j$, and $m(a) = j - i$ and $R_k(a) = \langle i, i+k-1 \rangle, \langle i+k, j \rangle$ when $j > i$

then the above recursive definition can be rewritten:

\[
\begin{align*}
\text{if } a &= \lambda \\
f(a) &= 0 \\
f(a) &= \text{Min}(\{f(\alpha[\alpha_1 + R_k(\alpha_1)]) \text{ if } \alpha \neq \lambda \}) \\
\text{initially } a &= (\langle 1, n \rangle)
\end{align*}
\]

* Means remove first component of $a$
Now this definition fits $a_1$ of theorem 3.3 with · being addition and $\Xi$ being Min. These operations have the appropriate properties so applying the theorem we get the following definition in which we have resubstituted for $R_j(a_1)$ being equivalent to the previous definition, with $a = <i,j>$, $i, j \in N$ and $j > i$.

$$
g(a) = \begin{cases} 0 & \text{if } i = j \\ \min_{k=1}^m ((x_i \times c_{i+k-1} \times c_j) + g(<i,i+k-1>) + g(<i+k,j>)) & \text{if } j > i 
\end{cases}
$$

Initially $a = <1,n>$

If we replace $g(<i,j>)$ by $X_{ij}$ and $x_i \times c_{i+k-1} \times c_j$ by $a_{ijk}$ in the above definition then that definition amounts to a set of $n^2$ equations, each equation being of the form:

$$X_{ij} = \min(a_{ij1} + X_{ii} + X_{i+1j} + a_{ij2} + X_{i1i+1}, \ldots)$$

This set of equations, like that for maximum path in a graph with no loops, forms the basis of a set $E_n$ that is open-loop consistent. The reason is that no equation in the equation set $E_n$ will have the same variable on both sides of the equation. This is because in the basis equations when $X_{ij}$ appears on the left and $X_{kp}$ appears on the right of an equation $j-i > k-p$, and so the same variable cannot appear on the right as on the left of any equation in $E_n$ since this property will be preserved even after substitution.

A process analogous to that described for maximum path can be used. The values of $X_{ij}$ with $i = j$ is known. By substituting these values, the solution of all $X_{ij}$ with $j-i = 1$ can be found. In general $X_{ij}$ with $j-i = k$ can be determined from the solutions for $X_{ij}$, $j-i = k+1$. 
until $X_{in}$ is found. This process has a complexity of $n^3$, the order of the number of equations times the maximum number of variables that appear on the right of an equation.

Conclusions:

The study of the relation of recursive definitions to their 'good' implementation is a place in which many important concepts and results, developed in diverse regions of computer science, seem to come together. One has here a natural way of classifying algorithms according to properties of their recursive definitions which cuts across relatively superficial classifications according to applications areas. The strong analogy between recursive definitions and differential equations on the one hand and 'good algorithms' and closed (or otherwise good) solutions on the other hand, supports the expectations that this study is a place to bring it all together. Well - almost all.
References:


Appendix I

Summary of Frequently used Notation

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

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

\( \mathbb{N}_\ast \) 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

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

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

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

\( \{A\} \) 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

\( E[x + y] \) 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 \( E[x + y, \; z + w] \) is the expression which results when each \( x \) is replaced by \( y \) and each \( z \) by \( w \) in \( E \).
An entire set of replacements can also be specified, i.e. if \( E, x_i \) and 
\( y_i \) for all \( i \in \mathbb{N} \) are expressions

\[
E'[x_i + y_i | i \in \mathbb{N}] \text{ is the expression that results when each occurrence of } x_i \text{ is replaced by } y_i \text{ in } E \text{ for all } i \in \mathbb{N}
\]

The notation also composes to allow specification of multiple replacements, i.e.

\[
(E[x + y])[x_i + y_i | i \in \mathbb{N}] = E'[x_i + y_i | i \in \mathbb{N}]
\]

where \( E' = E[x + y] \).

The notation is also extended to specify replacement of 1 component of \( E \).

Thus if \( E \) and \( x \) are expressions and \( i \) an integer

\( E[E_i + x] \) is the result of replacing the \( i^{th} \) symbol in \( E \) by the expression \( x \)

\( E[E_i+x+k] \) is the result of replacing the sub expression of the \( i^{th} \) thru \( i+k^{th} \) symbols in \( E \) with \( x \).

A further extension allows specifying the insertion of a string between symbols

\( E[E_i + x] = E[E_{(i+1)} x] \) is the result of inserting \( x \) between the \( i^{th} \) and \( i+1^{st} \) symbol of \( E \).