Eliminating Redundant Recursive Calls

NORMAN H. COHEN
Harvard University

The well-known recursive procedures to compute a given element of the Fibonacci series, to compute a binomial coefficient, and to solve the Towers of Hanoi puzzle define redundant computations. An invocation generally leads to many recursive calls with the same argument value. Such a redundant recursive procedure can be transformed into a nonredundant one when the operations appearing in the procedure have certain algebraic properties. The transformed programs avoid redundancy by saving exactly those intermediate results that will be needed again later in the computation.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.3 [**Programming Languages**]: Language Constructs—*procedures, functions and subroutines*; D.3.4 [**Programming Languages**]: Processors—*optimization*; *preprocessors*; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*program and recursion schemes*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*combinatorial algorithms*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*dynamic programming*

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: Redundant computation, recursive programs, recursion removal, pebbling, space bounds, program improvement

1. INTRODUCTION

Functions often have elegant definitions that are interpretations of the recursion schema $\mathcal{S}_n$, defined by

$\mathcal{S}_n$: **procedure** $f(x)$;
  value x; **anytype** x;
  if $p(x)$
    then $f := a(x)$
    else $f := b(x, f(c_1(x)), \ldots, f(c_n(x)))$.

However, many of these definitions are inefficient because they lead to redundant

A preliminary report on part of this research was presented at the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, January 1979 [7].
This work was supported by an IBM research fellowship and by the U.S. Department of the Navy under Naval Electronic Systems Command contract N00039-78-G-0020. The paper was revised while the author was employed by Sperry Univac.
Author's present address: Softech Inc., 705 Masons Mill Business Park, 1855 Byberry Road, Huntingdon Valley, PA 19006.

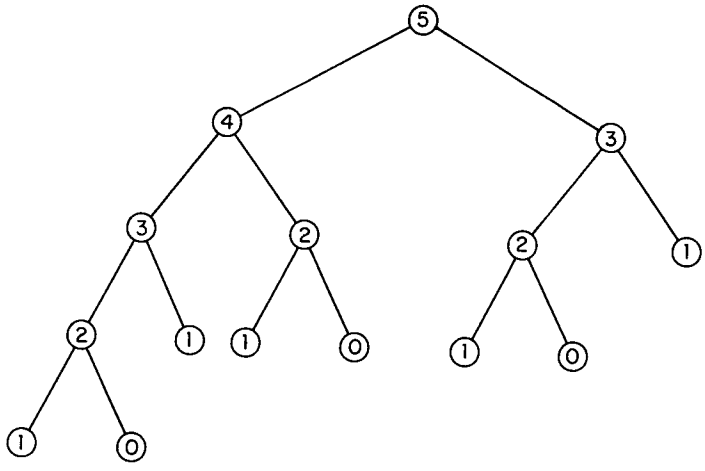


Fig. 1. A tree representation of the computation of $f(5)$ under the Fibonacci interpretation of $\mathscr{S}_2$. Each node represents a call on f and is labeled with the argument value for that call. The children of a node represent recursive calls arising directly from the call represented by that node.

computation. That is, they cause f to be called recursively several times with identical arguments.

For instance, one interpretation of $\mathscr{S}_2$ is the following function, which, given a nonnegative integer x, returns the xth element of the Fibonacci series:

```
integer procedure f(x);
  value x; integer x;
  if x = 0 or x = 1
    then f := 1
    else f := f(x − 1) + f(x − 2)
```

The resulting computation for $f(5)$ is pictured in Figure 1. Note that $f(3)$ is computed two times, and $f(2)$ is computed three times. In all, there are fifteen calls on f, with only six distinct argument values.

For certain interpretations of $\mathscr{S}_n$, transformations exist that, when applied to the procedure f, yield a nonredundant program that is partially equivalent to f. The applicability of these transformations depends on the validity of two conditions: a *descent condition* relating the functions c_i, $1 \leq i \leq n$, to each other and a *frontier condition* relating the c_i to the predicate p and the function a. For the Fibonacci interpretation, such a transformation would yield the following nonredundant program:

```
integer procedure f(x);
  value x; integer x;
  begin
    integer BACK0, BACK1, BACK2;
    f2(x);
    f := BACK0;
    procedure f2(x);
      value x; integer x;
```

```
    if x = 0 or x = 1
      then begin
             BACK0 := 1;
             BACK1 := 1
           end
      else begin
             f2(x − 1);
             BACK2 := BACK1;
             BACK1 := BACK0;
             BACK0 := BACK1 + BACK2
           end
end
```

The recursive procedure $f2(x)$ is executed for the side effect of placing the value of $f(x)$ (as originally defined) in $BACK0$. (It also places the value of $f(x-1)$ in $BACK1$ and the value of $f(x-2)$ in $BACK2$ for $x > 1$.)

We present four transformations, each applicable under a weaker descent condition than the previous one. We explain these transformations in terms of the schema $\mathcal{S} = \mathcal{S}_2$, defined by

```
S: anytype procedure f(x);
     value x; anytype x;
     if p(x)
       then f := a(x)
       else f := b(x, f(c(x)), f(d(x))).
```

(For simplicity we use c and d in place of c_1 and c_2.) We explore the constraints that each descent condition places on the domain and present a program that, given the validity of an appropriate frontier condition, exploits these constraints to compute f without redundancy. The solutions for $\mathcal{S}_2$ can be generalized to $\mathcal{S}_n$, $n \geq 2$. This generalization is detailed in [6, sec. IV.G.2].

We are interested both in exploration of the mathematical nature of redundant programs and in the practical improvement of redundant programs. After we consider the mathematical implications of each descent condition and present a transformed program reflecting the full generality of the schema $\mathcal{S}$, we discuss more specialized improvements that result in simpler and more efficient programs.

1.1 Assumptions and Terminology

It is assumed throughout that evaluating a, b, c, d, or p entails no side effects. The function b is assumed to be strict, so that call by value will compute the least fixpoint of the program f [18]. We guarantee that the transformed function returns the same value as f throughout the domain of f, but we do not attempt to characterize the behavior of the transformed function on those values for which f is not defined. The parameter x may be regarded either as a scalar or as a k-tuple $\langle x_1, \ldots, x_k \rangle$. In the latter case, the descent functions will be k-tuple-valued. We denote the set of natural numbers by N.

1.1.1 *Descent Trees and Compressed Descent DAGs.* A *descent tree* is a tree representing the computation of $f(x)$ for some x. Each node corresponds to a call on f: The root corresponds to the original call, and each node has children corresponding to the calls arising from the call on f represented by that node. Sometimes we identify nodes with the argument values of the corresponding

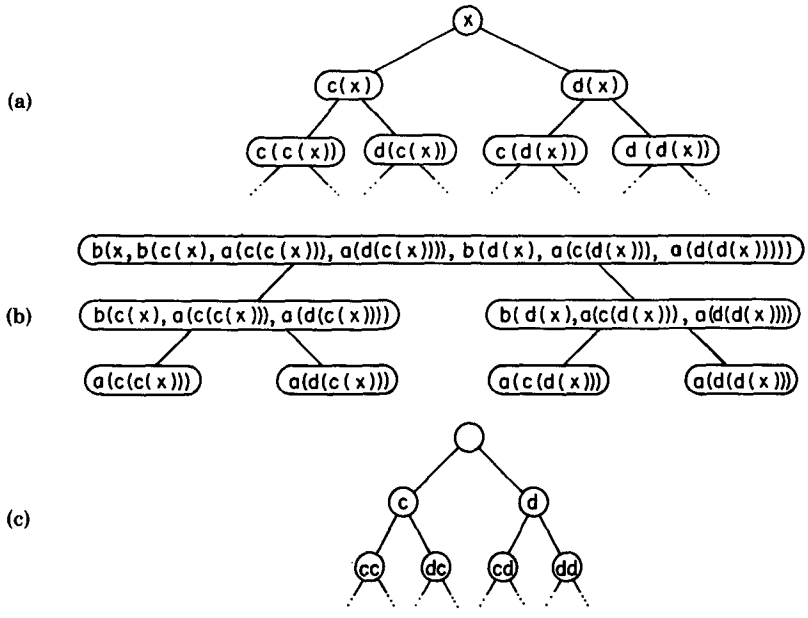


Fig. 2. Descent trees for the computation of $f(x)$. Tree (a) has nodes labeled with argument values, tree (b) with result values (under the assumption that p is true for every value in the third level of tree (a) and false for every value in the first and second levels of that tree). Tree (c) is a convenient abbreviation for tree (a).

calls, and sometimes we identify them with the result values of the corresponding calls. In the former case, we often abbreviate an argument value by indicating only a sequence of descent functions that must be applied to x to obtain that value. Each of these conventions is depicted in Figure 2. The tree in Figure 1 is a descent tree for the computation of $f(5)$, where f is the Fibonacci function.

If we make certain assumptions about c and d, then we may compress the descent tree into a directed acyclic graph (DAG) by merging certain (but not necessarily all) nodes corresponding to calls with the same argument value. (Any two such nodes are roots of identical subtrees, and we merge the nodes by merging those subtrees.) The resulting graph is called a *compressed descent DAG*. Figure 3 shows a compressed descent DAG for the descent tree of Figure 1. There can be many compressed descent DAGs for a given descent tree.

When we have merged *all* the nodes of a descent tree that can be shown to have the same value under a given set of assumptions, we call the resulting graph the *minimal* compressed descent DAG for that set of assumptions. Formally, if there is an interpretation obeying a given set of assumptions for which no two nodes of a compressed descent DAG have the same value, the compressed descent DAG is minimal. The minimal compressed descent DAG of a computation of $f(x)$ is uniquely defined.

1.1.2 *Programming Language.* Our programs and schemata are written in an extended version of ALGOL 60. The extensions are the data type **anytype** (corresponding to the uninterpreted domain of a schema); the data type **list**,

Fig. 3. A compressed descent DAG for the computation of $f(5)$, where f is the Fibonacci function. (Since each of the nodes has a different value, this is the minimal compressed descent DAG for that computation.)

together with list constants and list operations; the data type **string**, together with the string concatenation operator "‖"; and the arithmetic operator $(+_i)$, denoting addition modulo i where i is an integer constant.

1.2 Previous Approaches to the Problem

As in *dynamic programming* [1], we avoid redundant computation by saving results that may be needed many times. Michie [13] proposes implementing recursively defined functions as *memo functions* consisting of a *rote part* (a large table) and a *rule part* (a procedure for calculating values to be placed in the table). To evaluate a function, one first looks up its value in the table; if no value is there, one calculates the appropriate value, places it in the table for future reference, and returns it. Most approaches to eliminating redundant recursive calls have been variations on this *large-table method.*

Friedman, Wise, and Wand [10] implement such a table by using a LISP interpreter that suspends evaluation of the arguments to CONS until their values are required. This interpreter allows the definition of infinite lists. (At any instant the representation of such a list consists of a finite chain of dotted pair cells, with the CDR field of the last cell pointing to a suspended form that defines the rest of the list, together with a record of the environment in which that form is to be evaluated.) An infinite list can be defined that represents a table of values of a recursively defined function. Recursive calls can then be replaced by table lookups: The first time a table entry is referenced, the interpreter "coerces" a suspended form to obtain a value of the function. Successive references to that table entry find the value in place of the suspended form.

This scheme is similar to the *delay rule* of Vuillemin [18]. The delay rule is a computation rule specifying that a formal parameter should be evaluated the first time by evaluating the corresponding actual parameter in the calling environment, as in call by name, and saving the value; but subsequent evaluations of the same formal parameter should simply fetch this stored value. Under call by name, a recursive call occurring as an actual parameter of another recursive call will be invoked repeatedly with the same arguments whenever the corresponding actual parameter is evaluated. The delay rule avoids these redundant invocations while retaining the least fixpoint semantics of call by name. However, the delay rule does nothing to avoid the kind of redundancy with which the present paper is concerned. In Vuillemin's words, "We should blame inefficiencies [like the redundancy in the recursive Fibonacci program] on the program, not the computation rule." (Redundancy arising from the choice of a computation rule is not an issue with the schema $\mathscr{S}_n$; we are concerned with the kind of redundancy that should be "blamed on the program.")

The large-table method is simple and elegant, but it has several potential drawbacks. These include profligate and inefficient use of storage, slow table access, and overhead of storage maintenance (such as garbage collection, rehashing, or copying of dynamic **own** arrays). The extent to which any of these difficulties arise depends, of course, on the implementation of the table and the domain of the keys.

These inefficiencies can be avoided by employing the *small-table method.* There are two versions of the small-table method. The *constant* small-table method depends on an analysis of the program to determine that there is only a small set of values that will ever be looked up in the table. A small table is filled once and for all at the beginning of the computation, in an order that avoids redundant computation. The *variable* small-table method takes advantage of the fact that many of the result values saved in large tables are never referenced again: The table holds only some small number of previously calculated values at any time, and the contents of the table change during the course of the computation.

Bird [3] demonstrates the constant small-table method. He transforms two string-processing functions, replacing stack operations by calls on a recursive subroutine. In both cases this subroutine takes a single integer argument that is guaranteed to be within a certain reasonably small range. This makes the recursive function amenable to *tabulation* in an array. By a clever ad hoc analysis, Bird is able to find an order in which array elements may be efficiently filled with result values, using the definition of the recursive subroutine. The main program can then be modified so that values of the recursive subroutine are tabulated once and for all at the beginning of the program, and calls on the recursive program are replaced by array references. Unfortunately, this strategy, while quite effective, is not widely applicable, and it requires a fresh in-depth analysis of each program to which it is applied.

Hilden [11] applies the variable small-table method to the efficient computation of a statistical function, the Wilcoxan–Mann–Whitney probability. This function is defined by an interpretation of $\mathscr{S}$ with the property that $c(d(d(c(x)))) = d(c(c(d(x))))$ for all x (which gives rise to redundant recursive calls). In a typical invocation, the function is called recursively over a range of argument values so

large that an exhaustive table would not be feasible. Thus, a small hash table is used to save certain previously computed values. Hilden explores various heuristic strategies for deciding whether a given computed value should be saved or thrown away, and which previously saved values may be discarded to make room for a new value. Typical strategies are to save results arising at deeper levels of recursion, to save results arising at shallower levels of recursion, to save the most recently computed result for which the argument has a given hash value, or to save results arising from left-hand recursive calls (those with argument $c(x)$); but the most successful heuristic takes advantage of the fact that $c(d(d(c(x)))) = d(c(c(d(x))))$ for all x to assign priorities to various argument values.

Various approaches to the use of tables to eliminate redundant recursive computation, including the approach presented here, are surveyed in [2].

1.3 Our Approach

Like Hilden, we use a variable small table to remember *some* of the previously computed results. However, our approach is not heuristic. We replace table entries in a systematic manner that guarantees that, for the most general interpretations satisfying the constraints under consideration,

(1) a result is placed in the table only if it will be needed again later (or, toward the end of the computation, if all results that will be needed again are already in the table and there is room left over);
(2) a result that will be needed later is always placed in the table; and
(3) a result in the table is not replaced if it will be needed again later.

For three of the forms of redundancy that we consider, the size of the table required does not depend on the initial argument values. In such cases it is possible to compute $f(x)$ in constant space. This is significant because it has been shown [14, 16] that there exist interpretations of $\mathscr{S}$ that cannot be computed using a fixed amount of storage. For the fourth form of redundancy that we consider, the computation of $f(x)$ does, in general, require an amount of storage that is dependent on the initial value of x.

Unlike Friedman, Wise, and Wand [10], and unlike Bird [3], we do not save results from one "top-level" invocation of f to another. Thus, if f is called repeatedly by a main program, exhaustive tabulation is more efficient in terms of time, but less efficient in terms of space, than the approach that we propose. In such cases a more appropriate strategy is to transform the calling program instead of, or in conjunction with, f.

Our approach is more analytic than Hilden's, but less analytic than Bird's. We believe that it is applicable to a wide range of programs and that it could be incorporated into an automatic program-improvement system. This is discussed more fully in Section 6.

1.3.1 *General Strategy.* For each form of redundancy we construct a tree representing a typical value of $f(x)$. We then merge those nodes of the tree that represent identical values, forming a directed acyclic graph. The determination that certain nodes in the tree represent the same value is based on the descent condition, which asserts that certain sequences of repeated applications of c and d produce identical values. The DAG is partitioned into an ordered set of rooted

subgraphs with the property that the values in one subgraph can be determined from the set of values in the next subgraph.

The transformed program, $f2$, uses a nonlocal variable (usually an array) to store the result values corresponding to one such subgraph. The form of $f2$ is as follows:

```
procedure f2(x);
  value x; anytype x;
  if p(x)
    then initialize the nonlocal variable to the result values of the subgraph whose root
           corresponds to x
    else begin
           call f2 recursively with an argument equal to the root of the next subgraph,
             thus placing the result values of the next subgraph in the nonlocal
             variable;
           use the values in the nonlocal variable to compute the result values corre-
             sponding to the current subgraph, and update the variable with those
             values
         end
```

We make the assumption that, if p is true for the argument corresponding to the root of a subgraph, then p is either true or undefined for the argument corresponding to any other node in that subgraph. This assumption is expressed in the frontier condition. In the base case of $f2$, the frontier condition allows the element of the nonlocal variable corresponding to some value y to be initialized to $a(y)$.

The recursion in $f2$ is *linear*. That is, each path through the transformed program contains at most one recursive call. Linear recursions can always be removed without a stack, often quite efficiently [5, 6, 9, 17]. In the Fibonacci example, elimination of the recursion yields this program:

```
integer procedure f(x);
  value x; integer x;
  begin
    integer BACK0, BACK1, BACK2, i;
    BACK0 := 1;
    BACK1 := 1;
    for i from 1 until x - 1 do
      begin
        BACK2 := BACK1;
        BACK1 := BACK0;
        BACK0 := BACK1 + BACK2
      end;
    f := BACK0
  end
```

1.3.2 *Descent Conditions.* We partition the domain of f into two subsets, $\mathscr{B}$ and $\mathscr{R}$. $\mathscr{B}$ consists of the *base* cases, those values in the domain of f for which p is true; $\mathscr{R}$ consists of the *recursive* cases, those values in the domain of f for which p is false.

We refer to c and d as the *descent functions* of $\mathscr{S}$. (In the general formulation, c_i is a descent function of $\mathscr{S}_n$ for $1 \leq i \leq n$.) A *descent condition* is a requirement that certain relationships hold among the descent functions on $\mathscr{R}$. In Section 2 we consider the very strong descent condition that $c(x) = d(x)$ for all x in $\mathscr{R}$. In Section 3 we require the existence of a function g and integers m and n such that

$c(x) = g^m(x)$ and $d(x) = g^n(x)$ for all x in $\mathscr{R}$. (The notation $g^i(x)$ means x if $i = 0$ and $g(g^{i-1}(x))$ otherwise.) In Section 4 our descent condition asserts that $c^n(x) = d^m(x)$ and $c(d(x)) = d(c(x))$ on $\mathscr{R}$ for some m and n, but we do not insist on the existence of g. The descent condition for Section 5 is the relatively weak requirement that $c(d(x)) = d(c(x))$ on $\mathscr{R}$. (We say that c and d *commute* on $\mathscr{R}$.) Each of the descent conditions introduced in Sections 3, 4, and 5 is implied by the descent condition introduced in the previous section. Not surprisingly, as we consider progressively weaker descent conditions, our transformations become more intricate.

1.3.3 *The Frontier Condition.* The purpose of the frontier condition is to assure that there is a "nice" boundary between $\mathscr{R}$ and $\mathscr{B}$ and that the descent functions map values monotonically toward "easier" cases. Stating this formally, let S be a set of functions. (Typically, these functions are compositions of descent functions.) The *frontier condition* for S is the requirement that, for all x in $\mathscr{B}$ and all h in S,

(1) $a(h(x))$ (and thus $h(x)$) is defined; and
(2) if $f(h(x))$ is defined, then $f(h(x)) = a(h(x))$.

(This is always the case, for example, when h maps $\mathscr{B}$ into $\mathscr{B}$, that is, when $p(x)$ implies $p(h(x))$ for all h in S.) This condition assures us that, whenever we are asked for the value of $f(h(x))$, where h is in S and x is in $\mathscr{B}$, we may provide the answer $a(h(x))$: The computation of $a(h(x))$ is guaranteed to terminate, and the answer provided is correct whenever the value we were asked for is, in fact, defined. Put another way, the frontier condition assures us that, if x is a base case, then $h(x)$ may be *treated* as a base case for all h in S.

2. EXPLICIT REDUNDANCY

2.1 Definition of Explicit Redundancy

An interpretation of $\mathscr{S}$ exhibits *explicit redundancy* if $c(x) = d(x)$ for all x in $\mathscr{R}$. Consider, for example, the domain of LISP lists. Let $p(x)$ be the predicate NULL(x); let $a(x)$ be the constant function whose only value is the one-element list (( )); let $b(x, y, z)$ = APPEND(y, DISTRIBUTE(CAR(x), z)); let $c(x) = d(x)$ = CDR(x). DISTRIBUTE is a function taking an atom and a list of sublists and returning the result of CONSing the atom onto each sublist. (For example, DISTRIBUTE('A, '(( )(B)(C D))) is equal to the list ((A)(A B)(A C D)).) This interpretation leads to the following explicitly redundant definition mapping sets, represented as lists, to their power sets:

```
list procedure f(x);
  value x; list x;
  if NULL(x)
    then f := '(( ))
    else f := APPEND(f(CDR(x)),
                     DISTRIBUTE(CAR(x), f(CDR(x))))
```

2.2 Solution to Explicit Redundancy

Explicit redundancy is a very simple form of redundancy and is very easy to eliminate. The following solution models the solutions to be presented later for more complicated forms of redundancy.

We transform f into a recursive procedure $f2(x)$ that does not return a result but always leaves the value of $f(x)$ in a nonlocal variable T. Then f simply calls $f2$ and returns the value left in T, as follows:

```
anytype procedure f(x);
  value x; anytype x;
  begin
    anytype T;
    f2(x);
    f := T;
    procedure f2(x);
      value x; anytype x;
      if p(x)
        then T := a(x)
        else begin f2(c(x)); T := b(x, T, T) end
  end
```

The transformation is valid without the imposition of a frontier condition. (Formally, we require that the frontier condition hold for all functions in the empty set.) Of course, d may be used in place of c in the transformed program if it is preferable to do so.

2.3 Transformed Example

Applied to the power set program, this transformation results in the following nonredundant program:

```
list procedure f(x);
  value x; list x;
  begin
    list T;
    f2(x);
    f := T;
    procedure f2(x);
      value x; list x;
      if NULL(x)
        then T := '(( ))
        else begin
               f2(CDR(x));
               T := APPEND(T, DISTRIBUTE(CAR(x), T))
             end
  end
```

2.4 Causes of Explicit Redundancy

Explicit redundancy need not arise from poor programming. It is likely to be found in mechanically generated programs, or in programs written to maximize clarity rather than efficiency. The program above could have been a mechanical translation from pure LISP—in which there is no notion of assignment—to an ALGOL-like language. The original LISP programmer could have introduced an auxiliary function

```
(LAMBDA (X Y)
    (APPEND Y (DISTRIBUTE (CAR X) Y)))
```

and called it with arguments x and (f (CDR x)) to avoid the redundant computation, since (f (CDR x)) would be evaluated once and bound to Y. The resulting program would have been less clear, however.

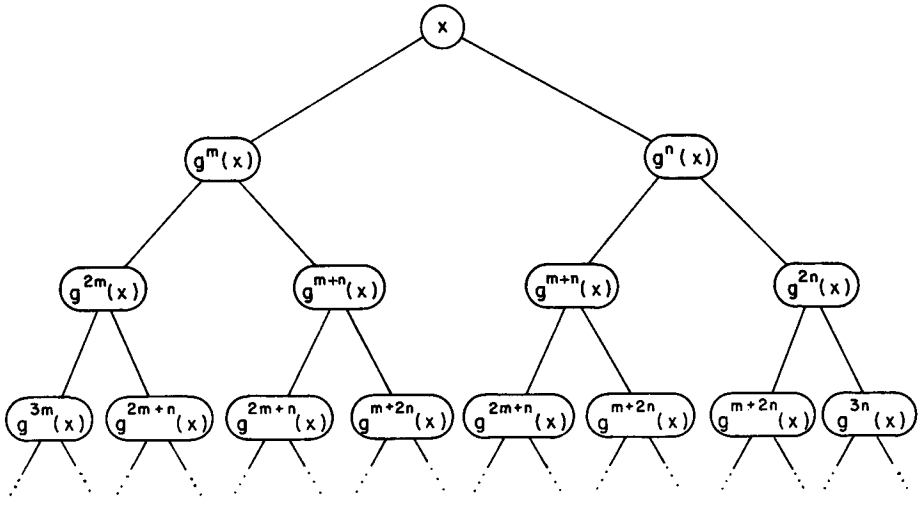


Fig. 4. Descent tree for common generator redundancy. The kth level of the tree has 2^{k-1} nodes but at most k distinct values.

3. COMMON-GENERATOR REDUNDANCY

3.1 Definition of Common-Generator Redundancy

Suppose that for a given interpretation of $\mathscr{S}$ there exist a function g and positive integers m and n such that $c(x) = g^m(x)$ and $d(x) = g^n(x)$ for all x in $\mathscr{R}$. We call g a *common generator* of c and d and say that the interpretation exhibits *common-generator redundancy*. We assume without loss of generality that m and n are relatively prime; if this is not the case, we consider $g^{\gcd(m,n)}$, $m/\gcd(m, n)$, and $n/\gcd(m, n)$ in place of g, m, and n, respectively, without affecting the value of g^m or g^n. Figure 4 shows the structure of the descent tree under these assumptions. In this section we present a transformation that eliminates common-generator redundancy, provided that the frontier condition holds for the set of functions $\{g^i \mid 0 < i \leq \max(m, n)\}$.

The Fibonacci program discussed in Section 1 exhibits common-generator redundancy. Recall that under that interpretation $c(x) = x - 2$ and $d(x) = x - 1$. If we define $g(x) = x - 1$, then $c(x) = g^m(x)$ and $d(x) = g^n(x)$ for $m = 2$, $n = 1$, and x in $\mathscr{R} = \{2, 3, \ldots\}$. The frontier condition requires that, for each function h in $S = \{\lambda x.x-1, \lambda x.x-2\}$ and each x in $\mathscr{B} = \{0, 1\}$, $f(h(x))$ is either undefined or equal to $a(h(x))$. The values of $h(x)$ for h in S and x in $\mathscr{B}$ are $0 - 1 = -1$, $0 - 2 = -2$, $1 - 1 = 0$, and $1 - 2 = -1$. The function a is a constant function, equal to 1 at all these points. Since $f(0) = 1 = a(0)$ and $f(-1)$ and $f(-2)$ are undefined, the frontier condition for S is satisfied.

3.2 Implications of Common-Generator Redundancy

The set of expressions occurring in the descent tree of Figure 4 is $\{g^{im+jn}(x) \mid i \in N, j \in N\}$. Since m and n are relatively prime, almost every natural number can be expressed in the form $im + jn$. Specifically, it is shown in [6, app. C] that all but $(m - 1)(n - 1)/2$ of the natural numbers can be so expressed; the highest

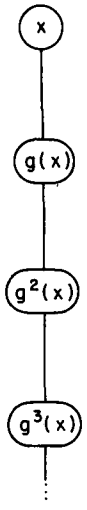


Fig. 5. The minimal compressed descent DAG for a computation of $f2(x)$ based on a recursive call $f2(g(x))$. The chain continues downward until, for some k, $p(g^k(x))$ is true.

number that cannot be is $(m - 1)(n - 1) - 1$. Typically, m and n will be small numbers.

Thus the tree contains expressions equivalent to almost every expression of the form $g^k(x)$, $k \in N$. This makes it reasonable to compute $f(x)$ by computing, in turn, each of the values $f(g^k(x)), f(g^{k-1}(x)), \ldots, f(g^1(x)), f(g^0(x)) = f(x)$ for some appropriate k. We shall write a procedure $f2(x)$ that calls itself recursively on $g(x)$ and then, using previous results, calculates the value of $f(x)$. The resulting descent tree for $f2$ is a simple chain, as shown in Figure 5. This tree is also a minimal compressed descent DAG.

Suppose we want to find the value of f at a given node in the chain, say that corresponding to $g^i(x)$, and $p(g^i(x))$ is false. If we look at the value computed m nodes down the chain, we find the value $f(g^{i+m}(x)) = f(g^m(g^i(x))) = f(c(g^i(x)))$. Similarly, the value computed n nodes down the chain is $f(d(g^i(x)))$. Using these, we can compute

$$b(g^i(x), f(c(g^i(x))), f(d(g^i(x)))),$$

which equals $f(g^i(x))$.

3.3 Solution to Common-Generator Redundancy

We use a nonlocal array $BACKg[0{:}max(m, n)]$ to remember values that may be needed later, and we transform f into a recursive procedure, $f2(x)$, that does not return a value but has the side effect of leaving the value of $f(g^i(x))$ in $BACKg[i]$, $0 \le i \le \max(m, n)$. (If $f(g^i(x))$ is undefined for some i, a meaningless value is left in $BACKg[i]$.) When $p(x)$ is true, the frontier condition assures us that either $f(g^i(x)) = a(g^i(x))$ or $f(g^i(x))$ is undefined, $0 \le i \le \max(m, n)$, so we may simply set $BACKg[i]$ to $a(g^i(x))$ for each i in this range. If $p(x)$ is false, we call $f2(g(x))$ to place $f(g^i(g(x))) = f(g^{i+1}(x))$ into $BACKg[i]$, $0 \le i \le \max(m, n)$, shift $BACKg[i]$ up to $BACKg[i + 1]$, $0 \le i \le \max(m, n) - 1$, and then place

$b(x, BACKg[m], BACKg[n]) = b(x, f(c(x)), f(d(x))) = f(x)$ in $BACKg[0]$. The program is as follows:

```
anytype procedure f(x);
  value x; anytype x;
  begin
    anytype array BACKg[0:(max(m, n))];
    f2(x);
    f := BACKg[0];
    procedure f2(x);
      value x; anytype x;
      if p(x)
        then begin
               integer i;
               for i from 0 until (max(m, n)) do
                 begin
                   BACKg[i] := a(x);
                   x := g(x)
                 end
             end
        else begin
               integer i;
               f2(g(x));
               for i from (max(m, n)) step -1 until 1 do
                 BACKg[i] := BACKg[i - 1];
               BACKg[0] := b(x, BACKg[m], BACKg[n])
             end
  end
```

Note that *BACKg* behaves as a queue: Newly calculated values are placed in *BACKg* at the low end of the array and migrate toward the high end as we ascend from the recursion. The values are referenced as they pass through the mth and nth positions, in order to calculate a new value to be placed at the beginning of *BACKg*. It is possible that a value will be calculated and placed in *BACKg* but never migrate far enough to be referenced, or that it will be referenced only to compute results that themselves never migrate far enough to be referenced, etc. However, the fact that all but $(m - 1)(n - 1)/2$ of the natural numbers can be expressed in the form $im + jn$, $i \in N, j \in N$, ensures that no more than

$$\max(m - 1, n - 1, (m - 1)(n - 1)/2)$$

useless values are computed for any x. (Any value $a(g^i(x))$ placed in *BACKg*, such that $f(g^i(x))$ is undefined, must turn out to be one of these useless values.)

3.4 Improving the Solution to Common Generator Redundancy

The queue *BACKg* can be implemented as a circular list by leaving elements of the queue stationary and advancing a pointer to the front of the queue. Recall, however, that m and n are constants whose values will, in practice, be small. Thus the **for** loops above may be unfolded into a sequence of assignments. Since array references in these assignments will have constant subscripts, we may replace *BACKg* by $\max(m, n) + 1$ scalar variables. Shifting each element of the queue then involves only $\max(m, n)$ scalar assignments, while a circular list requires

subscript calculations and code to handle wraparound. Unless m and n are unusually large, the circular list could well be *less* efficient.

Often there is a form that will evaluate directly to $a(g^i(x))$ in an amount of time independent of i. For example, if $g(x) = x - 1$, $a(g^i(x)) = a(x - i)$. In such cases we may replace the sequence "$BACKg[i] := a(x)$; $x := g(x)$" in the first **for** loop by the equivalent of "$BACKg[i] := a(g^i(x))$."

3.5 Transformed Example

The program resulting from elimination of redundancy in the Fibonacci program was presented in Section 1.

4. COMMUTATIVE PERIODIC REDUNDANCY[1]

4.1 The Impatient Commuter Problem

Consider the following shortest-path problem, which we shall call the Impatient Commuter Problem: There are two parallel highways into a city, and k interchanges at which a commuter may switch from one highway to the other. The time involved in switching is negligible, but all interchanges exit from the left lane and enter into the right lane. Since it takes time to cross several lanes of rush-hour traffic safely, we do not permit the commuter to switch highways at two consecutive interchanges. (That is, if a commuter switches highways at interchange i, he may not do so again until interchange $i + 2$.) The problem is to determine the minimum amount of time in which the commuter can reach the city, given the amount of time it takes to travel between each pair of consecutive interchanges on each highway.

Suppose we represent the highways by the integers 1 and 2 (so that, if we are on highway h, the other highway is $3 - h$), and we initialize the array $DELAY[0{:}k + 1, 1{:}2]$ so that $DELAY[i, h]$ is the time needed to travel between interchanges i and $i + 1$ on highway h, $1 \le i \le k$, and $DELAY[i, h] = 0$ for $i = 0$ and $i = k + 1$. (The interchange numbers are assumed to be in ascending order as we approach the city, and "interchange $k + 1$" is the final destination.) Then the following recursive program computes the solution to the Impatient Commuter Problem:

```
integer procedure f(i, h);
  value x; integer i, h;
  if i ≥ k
    then f := min(DELAY[i, h], DELAY[i, 3 − h])
    else f := min(DELAY[i, h] + f(i + 1, h),
                  DELAY[i, 3 − h]
                  + DELAY[i + 1, 3 − h]
                  + f(i + 2, 3 − h))
```

The value of $f(i, h)$ is the minimum time to reach the city if we are now approaching interchange i on highway h. If $i = k$, then we are at the last interchange, and we can simply select whichever highway takes us from inter-

[1] In a preliminary report on this work [7], commutative periodic redundancy was called simply "periodic redundancy." We now use the name "periodic redundancy" for another form of redundancy, described in Section 6.

change k to interchange $k + 1$ (the destination) most quickly. The time required to do this is

$$\min(DELAY[k, h], DELAY[k, 3 - h]).$$

If $i = k + 1$, we are already at the destination, so that the time required is

$$\min(DELAY[k + 1, h], DELAY[k + 1, 3 - h]) = \min(0, 0) = 0.$$

Otherwise, we take the smaller of the minimum time to reach the city if we stay on highway h and the minimum time to reach the city if we switch to highway $3 - h$. The first of these values is

$$DELAY[i, h] + f(i + 1, h);$$

the second is

$$DELAY[i, 3 - h] + DELAY[i + 1, 3 - h] + f(i + 2, 3 - h),$$

since the commuter is constrained to stay on highway $3 - h$ until at least the $(i + 2)$th interchange. The $(k + 1)$th row of $DELAY$ allows the base cases to be handled uniformly (by preventing the commuter from "overshooting" the city if he changes highways at intersection k); the zeroth row of delay allows us to assume without loss of generality that the commuter starts out at "interchange 0" on highway 1, so that we may solve the problem by calling $f(0, 1)$. (The commuter is allowed to switch highways at "interchange 0," so travel with nonzero cost may begin on either highway at interchange 1, with freedom to switch highways at interchange 2.)

The function f is, of course, an instance of $\mathscr{S}$, with x corresponding to the pair $\langle i, h \rangle$. Letting x_1 and x_2 denote the first and second components of the pair x,

$$\begin{aligned} p(x) &= (x_1 \geq k), \\ a(x) &= \min(DELAY[x_1, x_2], DELAY[x_1, 3 - x_2]), \\ b(x, y, z) &= \min(DELAY[x_1, x_2] + y, \\ &\qquad DELAY[x_1, 3 - x_2] + DELAY[x_1 + 1, 3 - x_2] + z), \\ c(x) &= \langle x_1 + 1, x_2 \rangle, \end{aligned}$$

and

$$d(x) = \langle x_1 + 2, 3 - x_2 \rangle.$$

The computation of f is redundant, as shown by the descent tree for $f(0, 1)$ with $k = 4$ in Figure 6. This is not surprising when we observe that $c^4(x) = d^2(x) = \langle x_1 - 4, x_2 \rangle$ and $c(d(x)) = d(c(x)) = \langle x_1 + 3, 3 - x_2 \rangle$.

4.2 Definition of Commutative Periodic Redundancy

In general, we say that an interpretation of $\mathscr{S}$ exhibits *commutative periodic redundancy* whenever there exist integers i and j such that $c^i(x) = d^j(x)$ and $c(d(x)) = d(c(x))$ for all x in $\mathscr{R}$. This is a generalization of common-generator redundancy: If c and d have a common generator g such that $c(x) = g^m(x)$ and $d(x) = g^n(x)$ on $\mathscr{R}$, then $c^n(x) = d^m(x) = g^{mn}(x)$ and $c(d(x)) = d(c(x)) = g^{m+n}$ on

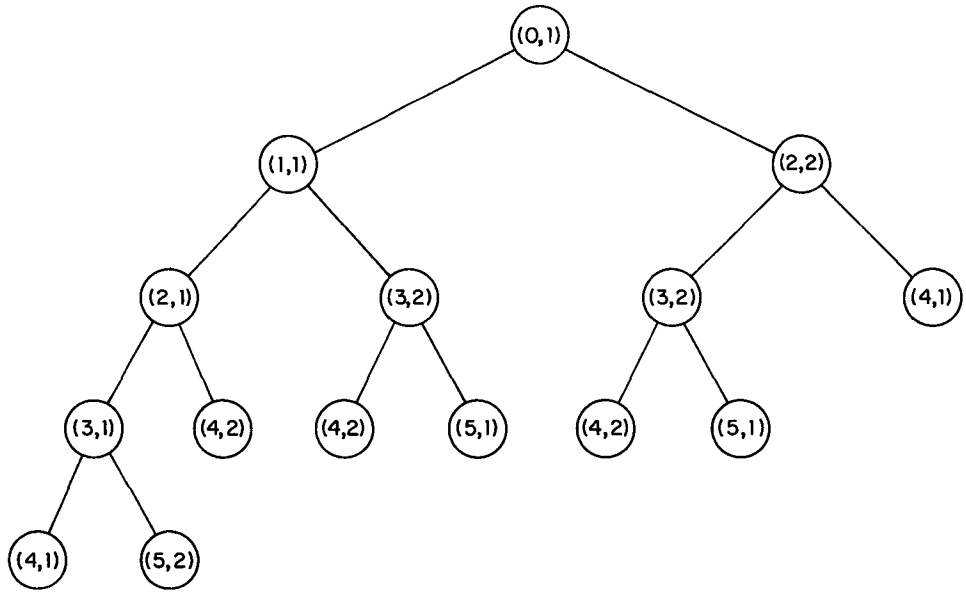


Fig. 6. Descent tree for the call $f(0, 1)$, where f is the program computing the solution to the Impatient Commuter Problem for $k = 4$. Each node is labeled with a pair (i, h) representing the argument values at that call.

$\mathscr{R}$. However, c and d in the solution to the Impatient Commuter Problem have no common generator. Thus the generalization is strict. We present a transformation for programs exhibiting commutative periodic redundancy that is valid whenever the frontier condition holds for the set of functions $\{c^m d^n \mid 0 \le m < i, 0 \le n < j, m + n > 0\}$.

4.3 Implications of Commutative Periodic Redundancy

Let $e = c^i = d^j$ on $\mathscr{R}$. Since c and d commute, all values in the minimal compressed descent DAG of $f(x)$ are of the form $c^m d^n(x)$ where m and n are nonnegative integers. But m can be expressed as $ri + s$ where r and s are nonnegative integers and $s < i$, and n can be expressed as $tj + u$ where t and j are nonnegative integers and $u < j$. Then $c^m d^n = c^{ri+s} d^{tj+u} = e^r c^s e^t d^u = e^{r+t} c^s d^u$. Thus each argument value in a compressed descent DAG for $f(x)$ has a unique representation of the form $e^v c^s d^u$ where $s < i$ and $u < j$.

Consider the subgraph of the compressed descent DAG consisting of the values $e^k c^s d^u$, $0 \le s < i$, $0 \le u < j$, for a given k. This subgraph is depicted in Figure 7. The subgraph forms an $i \times j$ rectangle. One can envision a comparable $i \times j$ rectangle whose upper right-hand side lies adjacent to the lower left-hand side of the depicted rectangle, and a third $i \times j$ rectangle whose upper left-hand side lies adjacent to the lower right-hand side of the depicted rectangle. In fact, the entire descent DAG may be compressed into a network of rectangles, as shown in Figure 8. Given result values corresponding to the upper sides of, for instance, rectangles D and E, it is possible to compute the result values corresponding to rectangle B.

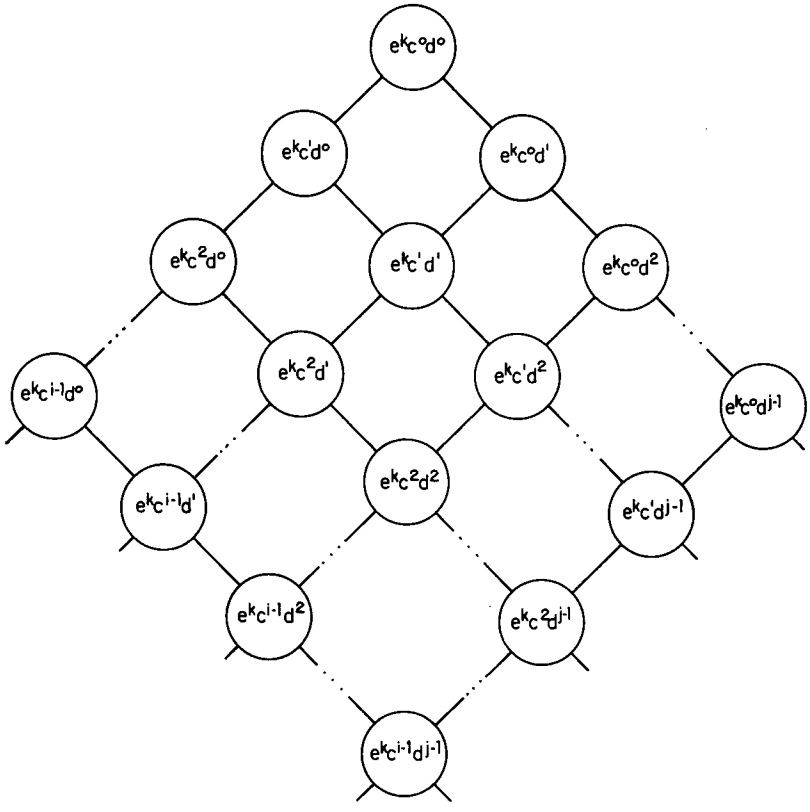


Fig. 7. The subgraph of a compressed descent DAG consisting of the nodes $e^kc^sd^u$, $0 \le s < i, 0 \le u < j$.

Let us assume that the rectangle labeled A represents the subgraph shown in Figure 7. The node in the uppermost corner of B is the one just below and to the left of the node in A corresponding to $e^kc^{i-1}d^0$; thus, it corresponds to $c(e^kc^{i-1}d^0) = e^kc^id^0$. The node in the uppermost corner of C occurs just below and to the right of the node in A corresponding to $e^kc^0d^{j-1}$; thus, it corresponds to $d(e^kc^0d^{j-1}) = e^kc^0d^j$.

However, $c^i = d^j = e$ on $\mathcal{R}$. That is, the nodes in the uppermost corners of rectangles B and C both correspond to $e^{k+1}c^0d^0$. (These rectangles would not be part of the descent DAG if $e^kc^{i-1}(x)$ and $e^kd^{j-1}(x)$, respectively, were not in $\mathcal{R}$.) It follows that all the nodes in B correspond to the same values as their counterparts in C. In other words, rectangles B and C are coextensive. Similarly, rectangles D, E, and F are mutually coextensive. Thus the descent DAG may be further compressed into the form shown in Figure 9. This compression preserves connections between rectangles; that is, if there is an edge between a certain node in rectangle x and a certain node in rectangle y in Figure 8, there is a connection between the corresponding nodes in the representatives of rectangles x and y in Figure 9.

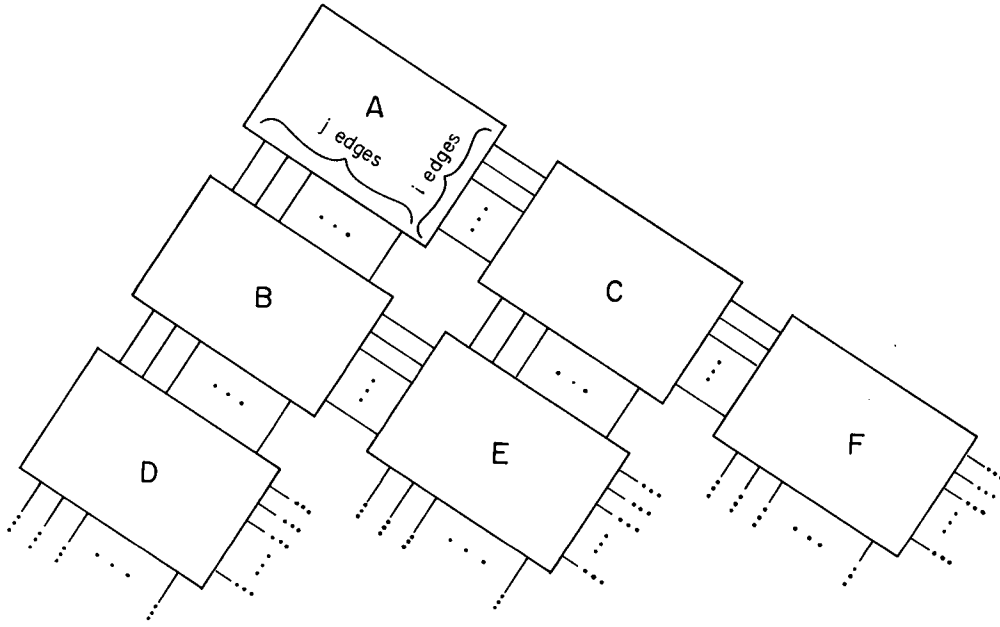


Fig. 8. An abstraction of the descent DAG. Each rectangle represents $i \times j$ nodes arranged in the configuration of Figure 7.

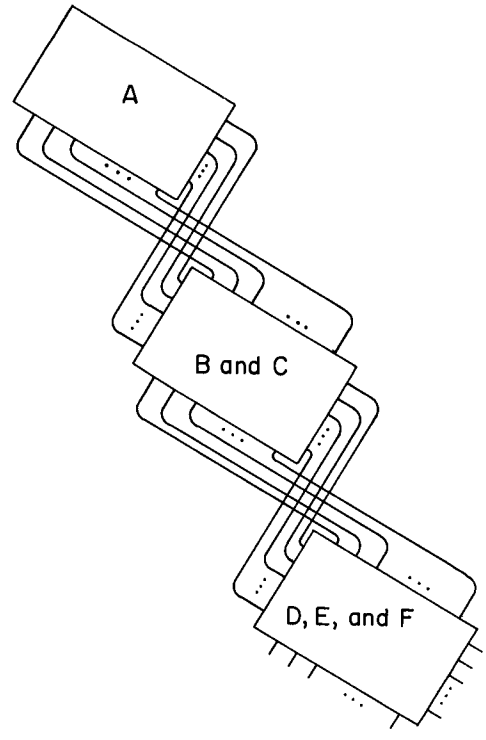


Fig. 9. The result of compressing the descent DAG by merging identical rectangles.

Each rectangle in Figure 8 has an uppermost corner corresponding to a value of the form $e^k c^0 d^0$. Every other node in that rectangle corresponds to some value $e^k c^s d^u$, where s and u are uniquely determined by the position of the node within the rectangle. There is a linear ordering among the rectangles in the descent DAG of Figure 9, based on the various rectangles' values for k. (Each value of k corresponds to a level of the tree.) Therefore, the compressed descent DAG of Figure 9 is minimal under the given assumptions.

4.4 Solution to Commutative Periodic Redundancy

These observations lead us to the following method for computing the value of $f(x)$ without redundancy: We maintain a nonlocal array, $RESULT[0:i-1, 0:j-1]$. We transform f into a procedure $f2(x)$ with the property that a call on $f2(x)$ leaves the value of $f(c^s d^u(x))$ (if that value is defined) in $RESULT[s, u]$, $0 \le s < i$, $0 \le u < j$. If $p(x)$ is true, this can be done immediately, according to the frontier condition, by setting $RESULT[s, u]$ to $a(c^s d^u(x))$. Otherwise, we call $f2$ recursively on $e(x) = c^i(x) = d^j(x)$, which has the effect of initializing $RESULT$ to the values appropriate for the next lower rectangle in the compressed descent DAG. Given these values, $f2$ can set $RESULT$ to the appropriate values for the current rectangle as follows (where $(+_i)$ and $(+_j)$ represent addition modulo i and modulo j, respectively):

```
begin
  anytype array ARGUMENT[0:(i − 1), 0:(j − 1)];
  integer M, N;
  for M from 0 until (i − 1) do
    begin
      ARGUMENT[M, 0] :=
        if M = 0 then x else c(ARGUMENT[M − 1, 0]);
      for N from 1 until (j − 1) do
        ARGUMENT[M, N] := d(ARGUMENT[M, N − 1])
    end;
  for M from (i − 1) step −1 until 0 do
    for N from (j − 1) step −1 until 0 do
      RESULT[M, N] :=
        if p(ARGUMENT[M, N])
          then a(ARGUMENT[M, N])
          else b(ARGUMENT[M, N],
                 RESULT[M (+i) 1, N],
                 RESULT[M, N (+j) 1])
end
```

The loop assigning a new value to $RESULT[M, N]$ proceeds backward through each row of $RESULT$ and, within each row, backward through each column. The assignments to the last row reference the values left in the first row by the recursive call. These values are replaced during the last iteration of the outer loop. Within each row, the assignment to the last column references the value left in the first column by the recursive call. That value is replaced during the last iteration of the inner loop. (The parenthesized expressions $(i - 1)$ and $(j - 1)$ can be replaced by constants for a given interpretation. Of course, the roles of c and d are interchangeable, simply by interchanging the roles of i and j.)

4.5 Improving the Solution to Commutative Periodic Redundancy

A slightly less clear version of this program uses $i + j$ locations to store intermediate results, instead of the $i \times j$ locations in *RESULT*. Observe that the only locations in *RESULT* that are important either at the beginning or the end of the second "**for** *M*" loop are those in row zero and column zero of *RESULT*. We introduce nonlocal vectors $BACKc[0:i-1]$ to represent column zero of *RESULT* and $BACKd[0:j-1]$ to represent each row of *RESULT* in turn. (Just before the $M = q$ iteration of the loop below, *BACKd* will represent the $(q\ (+_i)\ 1)$th row of *RESULT*; just after that iteration, it will represent the qth row of *RESULT*.) In the improved version of the algorithm, the second "**for** *M*" loop is rewritten as follows:

```
for M from (i − 1) step −1 until 0 do
  begin
    BACKd[(j − 1)] :=
      if p(ARGUMENT[M, (j − 1)])
        then a(ARGUMENT[M, (j − 1)])
        else b(ARGUMENT[M, (j − 1)], BACKd[(j − 1)], BACKc[M]);
    for N from (j − 2) step −1 until 0 do
      BACKd[N] :=
        if p(ARGUMENT[M, N])
          then a(ARGUMENT[M, N])
          else b(ARGUMENT[M, N], BACKd[N], BACKd[N + 1]);
    BACKc[M] := BACKd[0]
  end
```

In addition, the array *ARGUMENT* can be eliminated if c and d have inverses. Let *cInv* and *dInv* be functions such that $cInv(c(x)) = x$ and $dInv(d(x)) = x$ for all x in the domain of c and d. If the statements initializing *ARGUMENT* are replaced by statements setting a new scalar variable *SaveArg* to the value of $c^{i-1}d^{j-1}(x)$, then the loop updating *BACKc* and *BACKd* can be rewritten as follows:

```
for M from (i − 1) step −1 until 0 do
  begin
    Arg := SaveArg;
    BACKd[(j − 1)] :=
      if p(Arg)
        then a(Arg)
        else b(Arg, BACKd[(j − 1)], BACKc[M]);
    for N from (j − 2) step −1 until 0 do
      begin
        Arg := dInv(Arg);
        BACKd[N] :=
          if p(Arg)
            then a(Arg)
            else b(Arg, BACKd[N], BACKd[N + 1])
      end;
    BACKc[M] := BACKd[0];
    SaveArg := cInv(SaveArg)
  end;
```

Alternatively, if the value of $c^M d^N(x)$ can be expressed in an explicit form that can be evaluated directly in an amount of time independent of M and N, it is not

necessary to maintain *SaveArg* and *Arg*: We simply replace assignments of the form

```
BACKd[n] :=
  if p(Arg)
    then a(Arg)
    else b(Arg, . . . , . . .)
```

by

```
T := (explicit form for c^M d^n(x));
BACKd[n] := if p(T) then a(T) else b(T, . . . , . . .)
```

When this improvement is possible, the final result takes on the following form:

```
anytype procedure f(x);
  value x; anytype x;
  begin
    f2(x);
    f := BACKc[0];
    procedure f2(x);
      value x; anytype x;
      if p(x)
        then begin
               integer M, N;
               for M from 0 until (i − 1) do
                 BACKc[M] :=
                   a((explicit form for c^M d^0(x)));
               for N from 0 until (j − 1) do
                 BACKd[N] :=
                   a((explicit form for c^0 d^N(x)))
             end
        else begin
               integer M, N;
               anytype T;
               f2(e(x));
               for M from (i − 1) step −1 until 0 do
                 begin
                   T := (explicit form for c^M d^(j−1)(x));
                   BACKd[(j − 1)] :=
                     if p(T)
                       then a(T)
                       else b(T, BACKd[(j − 1)], BACKc[M]);
                   for N from (j − 2) step −1 until 0 do
                     begin
                       T := (explicit form for c^M d^N(x));
                       BACKd[N] :=
                         if p(T)
                           then a(T)
                           else b(T, BACKd[N], BACKd[N + 1])
                     end;
                   BACKc[M] := BACKd[0]
                 end
             end
  end
```

(As before, the use of **for** loops and arrays is only a notational convenience. Loops can be replaced by a sequence of assignments to scalar variables, and we may have a different explicit form for each value $c^m d^n(x)$ occurring in that sequence.)

4.6 Transformed Example

As the Impatient Commuter Problem is described, the frontier condition does not hold for any of the functions $\{c^0d^1, c^1d^0, c^1d^1, c^2d^0, c^2d^1, c^3d^0, c^3d^1\}$. This is because f can be called recursively with its first argument as high as $k + 1$, and evaluating $a(g(\langle k + 1, h\rangle))$, where g is any one of these functions, results in a subscript going out of bounds. This is easily remedied by adding sentinel rows $k + 2$ through $k + 6$ to *DELAY*, with each element in these rows equal to zero.

In applying the transformation, it is convenient to retain the "**for** M" loops indexing *BACKc*, but to unfold the "**for** N" loops indexing *BACKd* and replace the array *BACKd* by scalars *BACKd*0 and *BACKd*1. (The second "**for** N" loop would only be executed for $N = 0$ anyway.) A further improvement can be obtained by examining the single recursive call $f2(i + 4, h)$ and deducing that the value of h in any invocation of $f2$ is identical to its value in the top-level invocation of $f2$. Since we intend to invoke $f2$ initially only with $h = 1$, we can drop the second argument to $f2$, replacing all occurrences of h and $3 - h$ by 1 and 2, respectively. This results in the following version of $f2$:

```
procedure f2(i);
  value i; integer i;
  if i ≥ k
    then begin
           integer M;
           for M from 0 until 3 do
             BACKc[M] := min(DELAY[i + M, 1], DELAY[i + M, 2]);
           BACKd0 := min(DELAY[i, 1], DELAY[i, 2]);
           BACKd1 := min(DELAY[i + 2, 2], DELAY[i + 2, 1])
         end
    else begin
           integer M;
           integer T;
           f2(i + 4);
           for M from 3 step −1 until 0 do
             begin
               T := i + M + 2;
               BACKd1 :=
                 if T ≥ k
                   then min(DELAY[T, 2], DELAY[T, 1])
                   else min(DELAY[T, 2] + BACKd1,
                            DELAY[T, 1] + DELAY[T + 1, 1] + BACKc[M]);
               T := i + M;
               BACKd0 :=
                 if T ≥ k
                   then min(DELAY[T, 1], DELAY[T, 2])
                   else min(DELAY[T, 1] + BACKd0,
                            DELAY[T, 2] + DELAY[T + 1, 2] + BACKd1)
               BACKc[M] := BACKd0
             end
         end
```

Since $DELAY[i + 2, 1] = DELAY[i + 2, 2] = 0$ whenever $i \geq k$, the first assignment to *BACKd*1 can be replaced by "*BACKd*1 := 0". A further improvement can be obtained by dividing the recursive case into two subcases, $k - 5 \leq$

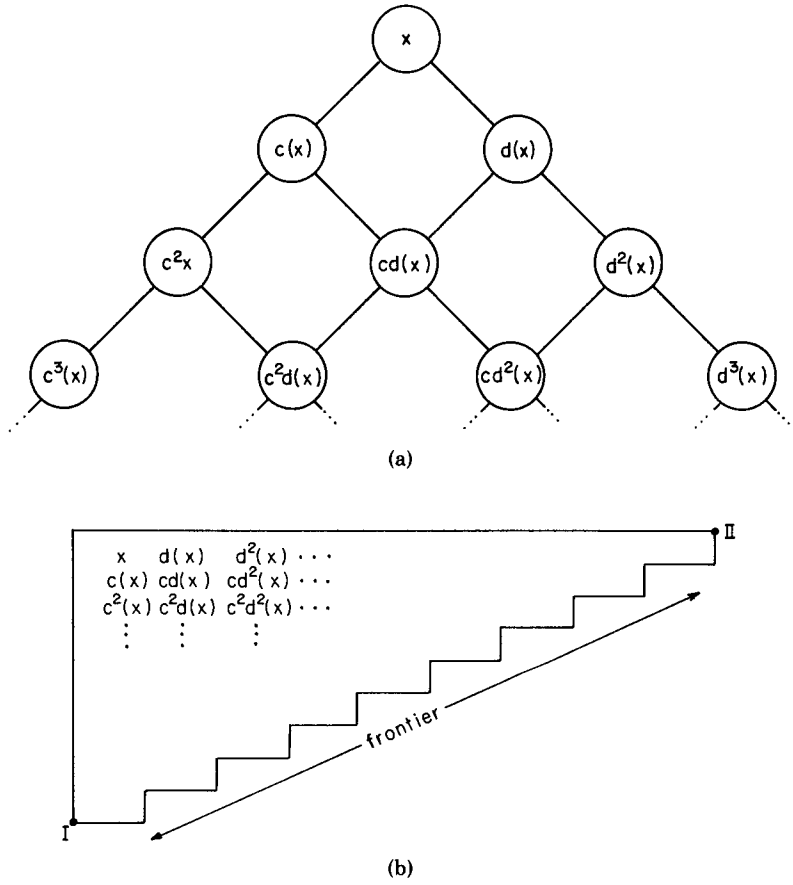


Fig. 10. The minimal compressed descent DAG for commutative redundancy. In (b), the DAG of (a) has been rotated 45° counterclockwise to illustrate the structure of the frontier whenever the frontier condition holds.

$i \leq k - 1$ and $0 \leq i \leq k - 6$. In the second subcase, the two "**if** $T \geq k$" statements in the **for** loop can be replaced by their respective **else** parts.

5. COMMUTATIVE REDUNDANCY

5.1 Definition of Commutative Redundancy

Finally, we consider interpretations for which the only descent condition is that $c(d(x)) = d(c(x))$ for all x in $\mathcal{R}$. This generalization of commutative periodic redundancy is called *commutative redundancy*. Figure 10a shows the minimal compressed descent DAG under this descent condition. Our transformation for eliminating commutative redundancy is valid whenever the frontier condition holds for the set of functions $\{c^i d^j \mid i \in N, j \in N\}$. Whenever this is the case, the DAG for the entire computation takes on the shape shown in Figure 10b (where the DAG has been turned on its side for clarity); that is, the frontier between $\mathcal{R}$

and $\mathscr{B}$ proceeds monotonically upward and rightward from point I to point II. Formally stated,

$$\min\{k \mid p(c^k(d(x)))\} \leq \min\{k \mid p(c^k(x))\}$$

and

$$\min\{k \mid p(d^k(c(x)))\} \leq \min\{k \mid p(d^k(x))\},$$

for all x in $\mathscr{R}$.

An example of commutative redundancy is the following recursive definition of the binomial coefficient of n and k, $n!/(k!(n-k)!)$, defined for $n \geq 0$ and $0 \leq k \leq n$:

```
integer procedure f(n, k);
  value n, k; integer n, k;
  if k = 0 or k = n
    then f := 1
    else f := f(n − 1, k − 1) + f(n − 1, k)
```

This is an instance of $\mathscr{S}$ with

$$x = \langle n, k \rangle,$$

$$p(\langle x_1, x_2 \rangle) = (x_2 = 0 \text{ or } x_2 = x_1),$$

$$a(x) = 1,$$

$$b(x, y, z) = y + z,$$

$$c(\langle x_1, x_2 \rangle) = \langle x_1 - 1, x_2 - 1 \rangle,$$

and

$$d(\langle x_1, x_2 \rangle) = \langle x_1 - 1, x_2 \rangle.$$

Since $c(d(\langle x_1, x_2 \rangle)) = d(c(\langle x_1, x_2 \rangle)) = \langle x_1 - 2, x_2 - 1 \rangle$, c and d commute on $\mathscr{R}$. For i and j in N, $c^i d^j(\langle x_1, x_2 \rangle) = \langle x_1 - i - j, x_2 - j \rangle$. Thus the frontier condition asserts that for all i and j in N, if $x_2 = 0$ or $x_2 = x_1$, the binomial coefficient of $x_1 - i - j$ and $x_2 - j$ is either equal to one or undefined. The proof of the frontier condition is straightforward.

5.2 Implications of Commutative Redundancy

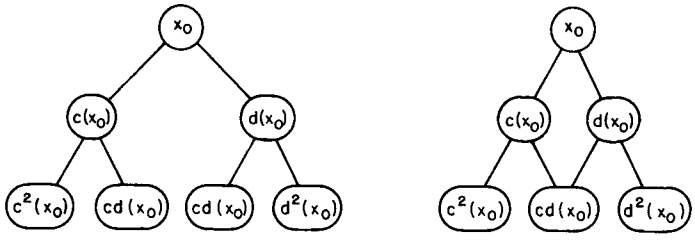
Paterson and Hewitt [14] showed that for each n there exist interpretations for the schema $\mathscr{S}$ for which the function f cannot be computed using fewer than n storage locations for partial results. An adaptation of their proof shows that this is true even if the interpretations are restricted to be commutatively redundant. Consequently, if we are to eliminate commutative redundancy by using a nonlocal array to store needed temporary results, we shall, in general, have to allocate that array dynamically.

The adapted proof differs from the original only in that it replaces a descent tree with the corresponding minimal compressed descent DAG. For each n, we consider the interpretation $\mathscr{I}_n$, which is free except for the assumptions that $c(d(x)) = d(c(x))$ whenever $p(x)$ is false and that $p(c^i d^j(x_0))$ is true if and only

$\mathcal{I}_0$: $f(x_0) = a(x_0)$

$\mathcal{I}_1$: $f(x_0) = b(x_0, a(c(x_0)), a(d(x_0)))$

$\mathcal{I}_2$: $f(x_0) = b(x_0, b(c(x_0), a(c^2(x_0)), a(cd(x_0))), b(d(x_0), a(cd(x_0)), a(d^2(x_0))))$



$\mathcal{I}_3$: $f(x_0) = b(x_0, b(c(x_0), b(c^2(x_0), a(c^3(x_0)), a(c^2d(x_0))), b(cd(x_0), a(c^2d(x_0)), a(cd^2(x_0)))),$
$b(d(x_0), b(cd(x_0), a(c^2d(x_0)), a(cd^2(x_0))), b(d^2(x_0), a(cd^2(x_0)), a(d^3(x_0)))))$

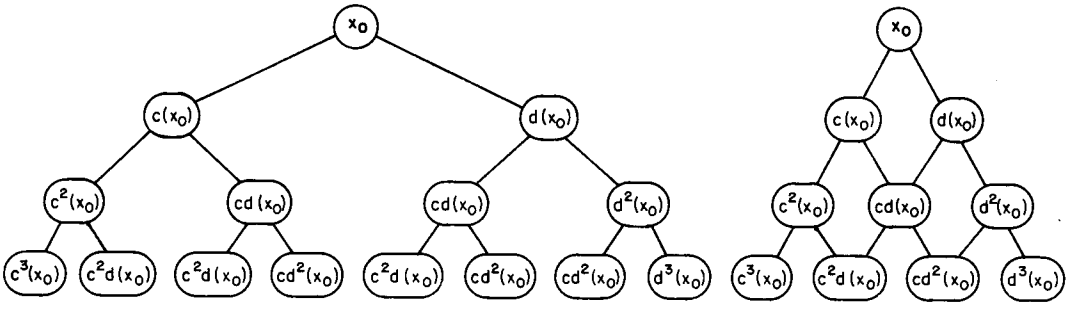


Fig. 11. The value of $f(x_0)$ under each of the interpretations $\mathcal{I}_0$, $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$. The descent tree and minimal compressed descent DAG are shown for each interpretation.

if $i + j \geq n$, for constant x_0. The value of $f(x_0)$, along with the descent trees and minimal compressed descent DAGs, is displayed in Figure 11 for interpretations $\mathcal{I}_0$, $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$.

We now show that the computation of $f(x_0)$ under interpretation $\mathcal{I}_n$ requires at least $n + 1$ storage locations. Each step of the computation involves either (1) placing the value corresponding to a leaf of the DAG in a storage location or (2) taking the values corresponding to children of a given interior node from two storage locations, computing the value corresponding to that interior node, and placing the computed value in a storage location. We call the DAG *open* if there is a path from the root to a leaf consisting only of nodes whose corresponding values are not currently held in any storage location. Otherwise, the DAG is *closed*. The computation begins with the DAG open and concludes with the DAG

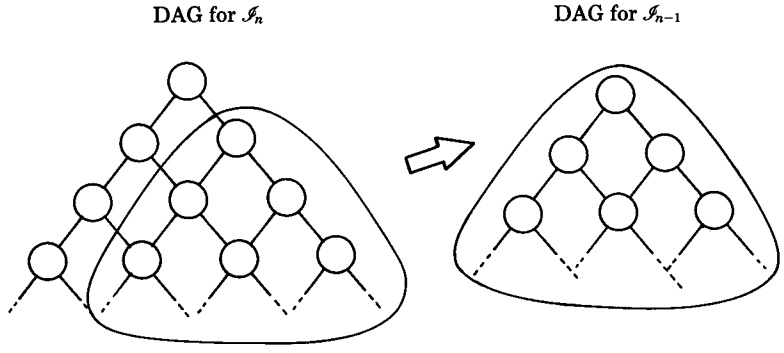


Fig. 12. A mapping from the DAG for $\mathscr{I}_n$ to the DAG for $\mathscr{I}_{n-1}$. The ith node on the kth level of the first DAG is mapped to the $(i-1)$th node on the $(k-1)$th level of the second DAG.

closed (with the value corresponding to the root in a storage location). Thus, it suffices to show that $n+1$ storage locations are required at the moment a DAG shaped like the minimal compressed descent DAG for $\mathscr{I}_n$ becomes closed.

We do this by observing that the computation of $f(x_0)$ under $\mathscr{I}_0$ trivially requires one storage location and proving that computation of $f(x_0)$ under $\mathscr{I}_n$ requires at least one storage location more than under $\mathscr{I}_{n-1}$. Consider the following isomorphic mapping from a subgraph of the DAG for $\mathscr{I}_n$ to the DAG for $\mathscr{I}_{n-1}$: The ith node on the kth level of the DAG for $\mathscr{I}_n$ is mapped to the $(i-1)$th node on the $(k-1)$th level of the DAG for $\mathscr{I}_{n-1}$, $1 < i \leq k$. (Speaking intuitively, this mapping creates a graph the same shape as the DAG for $\mathscr{I}_{n-1}$ by cutting off the nodes on the upper left-hand edge of the DAG for $\mathscr{I}_n$, as shown in Figure 12.) Now consider a sequence of operations that closes the DAG for $\mathscr{I}_n$ leaving the fewest possible storage locations in use at the moment the DAG is closed. We can map this sequence to a sequence of operations closing the DAG for $\mathscr{I}_{n-1}$ by ignoring operations involving the first node on any level of the DAG for $\mathscr{I}_n$ and applying the node mapping just described to all other operations. However, for the DAG for $\mathscr{I}_n$ to be closed, at least one value corresponding to the first node on some level of the DAG must be in a storage location; otherwise, the leftmost path in the DAG must be open. Since this node plays no role in the sequence of operations we have created to close the DAG for $\mathscr{I}_{n-1}$, there is a way for the DAG for $\mathscr{I}_{n-1}$ to become closed using one storage location less at the moment of closure than the minimum number of storage locations required by $\mathscr{I}_n$ at the moment of closure. That is, the minimum number of locations required by $\mathscr{I}_n$ at the moment the DAG becomes closed is one more than the minimum for $\mathscr{I}_{n-1}$. This establishes that there exist interpretations for $\mathscr{S}$ for which there is no way to calculate f using a fixed amount of storage, even though those interpretations exhibit commutative redundancy.

(Proofs such as this are often explained in terms of a game played with pebbles [15]. Each pebble corresponds to a storage location, and placing a pebble on a node of a DAG corresponds to computing and storing the value associated with that node. In order to do this, one must already have stored the values corre-

sponding to any children of that node. Thus, a pebble may be placed on a node only if each child of that node is already pebbled. A pebble may be removed from a node at any time. The object of the game is to place a pebble at the root of the DAG using as few pebbles as possible.)

5.3 Solution to Commutative Redundancy

Define the *d-depth* of x to be the smallest natural number i such that $p(d^i(x))$ is true. Clearly, *d-depth*(x) has a value whenever $f(x)$ is defined. This value can be computed by the program

```
integer procedure d-depth(x);
  value x; anytype x;
  begin
    integer i;
    i := 0;
    while not p(x) do
      begin x := d(x); i := i + 1 end;
    d-depth := i
  end,
```

or, depending on the interpretation of p and d, it may be computable directly. (For instance, if $d(x) = x - 1$, $p(x)$ is "$x = 0$," and $x \in N$, *d-depth*$(x) = x$.) Our solution to commutative redundancy uses a nonlocal array with *d-depth*(x) locations to compute $f(x)$. We do not specify how *d-depth*(x) is to be computed.

The solution is to think of the DAG as drawn in Figure 10b and let the nonlocal array *BACKd* represent the result values in a row of that rotated DAG. Another nonlocal array, *ARGUMENT*, will hold the argument values in that row. We transform f into a procedure $f2(x, y)$ that leaves the value of $f(d^i(x))$ in $BACKd[i]$, $0 \le i \le y$. When *d-depth*(x) is 0 (i.e., $p(x)$ is true), this is done simply by placing $a(d^i(x))$ in $BACKd[i]$, $0 \le i \le y$. (The frontier condition guarantees that this is valid.) Otherwise, we call $f2$ recursively with arguments $c(x)$ and *d-depth*$(x) - 1$, which places $f(d^i(c(x)))$ in $BACKd[i]$, $0 \le i <$ *d-depth*(x). We then set the array *ARGUMENT* so that $ARGUMENT[i] = d^i(x)$, $0 \le i <$ *d-depth*(x). Next, we place $a(d^i(x))$ in $BACKd[i]$ for $i =$ *d-depth*(x) and any i such that *d-depth*$(x) < i \le y$. Finally, we execute the loop

```
for i from d-depth(x) − 1 step −1 until 0 do
  BACKd[i] := b(ARGUMENT[i], BACKd[i], BACKd[i + 1]),
```

which leaves $f(d^i(x))$ in $BACKd[i]$, $0 \le i \le y$. (At the beginning of the $i = k$ passage through the loop, $BACKd[j] = f(d^j(x))$ for $j = k + 1$ and any other j such that $k < j \le y$, and $BACKd[j] = f(d^j(c(x))) = f(cd^j(x))$ for $0 \le j \le k$. Since $k <$ *d-depth*(x), $p(x)$ is false, so

$$\begin{aligned} &b(ARGUMENT[i], BACKd[i], BACKd[i + 1]) \\ &\quad = b(d^k(x), f(cd^k(x)), f(d^{k+1}(x))) \\ &\quad = b(d^k(x), f(c(d^k(x))), f(d(d^k(x)))) \\ &\quad = f(d^k(x)). \end{aligned}$$

Thus, after the $i = k$ passage through the loop, $BACKd[k]$ holds $f(d^k(x))$.)

The entire program looks like this:

```
anytype procedure f(x);
  value x; anytype x;
  begin
    integer D;
    D := d-depth(x);
    begin
      anytype array BACKd[0:D], ARGUMENT[0:D];
      f2(x, D);
      f := BACKd[0];
      procedure f2(x, y);
        value x, y; anytype x; integer y;
        if p(x)
          then begin
                 integer i;
                 for i from 0 until y do
                   begin
                     BACKd[i] := a(x);
                     x := d(x)
                   end
               end
          else begin
                 integer i, z;
                 z := d-depth(x);
                 f2(c(x), z - 1);
                 for i from 0 until max(y, z) do
                   begin
                     ARGUMENT[i] := x;
                     x := d(x)
                   end;
                 for i from z until max(y, z) do
                   BACKd[i] := a(ARGUMENT[i]);
                 for i from z - 1 step -1 until 0 do
                   BACKd[i] :=
                     b(ARGUMENT[i], BACKd[i], BACKd[i + 1])
               end
    end
  end
```

5.4 Improving the Solution to Commutative Redundancy

As before, this program can be improved in many cases: If d is invertible, or if $d^i(x)$ can be computed in an amount of time independent of i (for example, when $d(x) = x - 1$, $d^i(x) = x - i$), then *ARGUMENT* can be eliminated. If neither $d^i(x)$ nor *d-depth*(x) can be computed directly, then time would be saved by combining the initialization of *ARGUMENT* with the calculation of *d-depth*. However, this would be expensive in terms of space, since we would then have to make *ARGUMENT* local to *f2* to save it across recursive calls.

Of course, the roles of c and d can be interchanged if appropriate changes are made throughout the program. (For instance, $b(ARGUMENT[i], BACKd[i], BACKd[i + 1])$ would become $b(ARGUMENT[i], BACKc[i + 1], BACKc[i])$ and *d-depth* would be replaced by *c-depth*.) Such changes would result in a more space-efficient program if *c-depth*(x) were usually smaller than *d-depth*(x).

If the value of $a(x)$ is the same for all x in $\mathcal{B}$, then *BACKd* can be initialized with every element equal to that constant value. Then the two loops which set

elements of *BACKd* to $a(x)$ for some x can be eliminated. (This can be done because the array elements which are set by these loops are not referenced by the program before the loops are encountered.)

5.5 Transformed Example

This last improvement is applicable in the binomial coefficient example, since $a(x) = 1$ for all x. Since $d(\langle n, k\rangle) = \langle n-1, k\rangle$ and $p(\langle n, k\rangle) = (k = 0 \textbf{ or } k = n)$, *d-depth*$(\langle n, k\rangle)$ is the least natural number i such that $k = 0$ or $k = n - i$, that is, 0 if $k = 0$ and $n - k$ otherwise. In contexts where $\langle n, k\rangle$ is known not to be a base case, we may assume that *d-depth*$(\langle n, k\rangle) = n - k$. This, together with the fact that $c(\langle n, k\rangle) = \langle n-1, k-1\rangle$, points to further simplifications:

(1) The value of *d-depth*(x) remains constant for all invocations of $f2$ except the base-case invocation. Therefore, all references to *d-depth*(x) in $f2$ can be replaced by references to D (which is set to *d-depth*(x) by f before $f2$ is called); we can eliminate the second argument to $f2$, replacing occurrences of z and $max(y, z)$ by D.

(2) If $n \neq k$ initially, then $n \neq k$ for all invocations of $f2$. Thus we can handle the case where $n = k$ separately without calling $f2$ and assume that $n \neq k$ throughout $f2$. In particular, the test "$k = 0$ **or** $k = n$" can be simplified to "$k = 0$." Then n is never referenced within $f2$, so we may replace the parameter $\langle n, k\rangle$ by k.

These simplifications result in the following program:

```
integer procedure f(n, k);
  value n, k; integer n, k;
  begin
    integer D;
    D := n - k;
    begin
      integer array BACKd[0:D];
      if n = k
        then f := 1
        else begin
               integer i;
               for i from 0 until D do BACKd[i] := 1;
               f2(k);
               f := BACKd[0]
             end;
      procedure f2(k);
        value k; integer k;
        begin
          integer i;
          if k ≠ 0
            then begin
                   f2(k - 1);
                   for i from D - 1 step -1 until 0 do
                     BACKd[i] := BACKd[i] + BACKd[i + 1]
                 end
        end
    end
  end
```

In this program k is simply acting as a counter: The entire effect of the call $f2(k)$ is to execute the **for** loop in $f2$ k times. The action of the program is easily

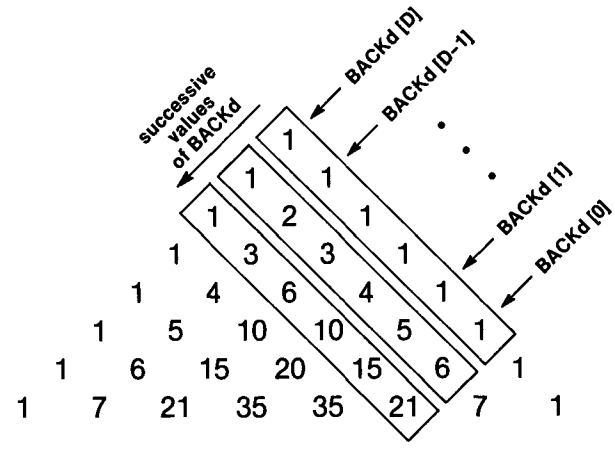


Fig. 13. The relationship of the transformed binomial coefficient program to Pascal's triangle. Initially, *BACKd* holds all ones. In ascending from the recursion the program computes values in successive parallel "slices" of the triangle. Ultimately, the binomial coefficient of n and k, which is the $(k + 1)$th value in the $(n + 1)$th row of the triangle, is left in *BACKd*[0].

understood in terms of Pascal's triangle: *BACKd* initially holds the first $n - k$ values along one of the edges of Pascal's triangle—all of which are 1. Each execution of the **for** loop computes the first $n - k$ values in the next diagonal parallel to this edge, using the additive rule for generating elements of Pascal's triangle. See Figure 13.

This program uses $\theta(k(n-k))$ time and $\theta(n-k)$ space. Because $f(n, k) = f(n, n - k)$ (a fact not obvious from the discussion above), space can be saved by replacing k with $n - k$ when $k < n/2$.

6. CONCLUSIONS

We have presented a recursion schema $\mathcal{S}$ defining a function f, together with various sets of assumptions under which the definition of f calls for redundant computation. For each set of assumptions we investigated the nature of the redundancy and presented a recursive program that computes f without redundant recursive calls. Weaker assumptions required more intricate computations and greater amounts of storage.

The use of nonlocal variables is only one of a number of implementations that could have been used. An obvious alternative would be to rewrite *f*2 to *return* a tuple consisting of those results that will be needed later, instead of leaving those values in nonlocal arrays. The nonlocal variables could also have been replaced by **own** (i.e., static) variables local to *f*2.

6.1 Automated Transformation Systems

The transformations presented here can be applied to actual programs. More important, they can be incorporated in automatic program-improvement systems. One such system is that of Burstall and Darlington [4]. That system uses heuristic methods to apply a series of simple but powerful transformations to recursively

defined functions. One of the most effective transformations is the *definition* of new functions in terms of old ones. Among the examples of definition presented by Burstall and Darlington is the definition of a function that returns a tuple consisting of the nth and $(n-1)$th elements of the Fibonacci series by calling itself recursively to obtain the values of the $(n-1)$th and $(n-2)$th elements of the series. This is exactly the program that would result from applying our transformation for common-generator redundancy to the usual recursive Fibonacci program, provided that we returned tuples instead of setting nonlocal arrays. This transformation originally required human intervention to provide the new function definition, but the formulation of the definition has since been automated [8] for programs exhibiting common-generator redundancy and having the common generator itself as one of the descent functions. The methods of this paper may be useful in generalizing this facility, providing both a way to recognize situations in which definition would be useful and a way to formulate the definition itself.

Effective procedures exist to find true descent conditions, given "blocks" of straight-line code that compute descent functions. Let $u_1, \ldots, u_q$ and $v_1, \ldots, v_r$ be blocks; let $b_1 b_2$ represent the block obtained by executing block b_1 and then using its output values as input values to block b_2; and let $b^n = bb^{n-1}$ for $n > 1$, with $b^1 = b$. Lewis [12] proves that it is possible to compute those values $i_1, \ldots, i_q$ and $j_1, \ldots, j_r$ for which

$$u_1^{i_1} \cdots, u_q^{i_q}$$

computes the same function as

$$v_1^{j_1} \cdots v_r^{j_r}.$$

The proof is by reduction to Presburger arithmetic. The development of practical algorithmic or heuristic procedures for finding true descent conditions requires further research.

6.2 Other Forms of Redundancy

Each form of redundancy we have discussed involves descent functions that commute. Redundant calls arise because arguments are just repeated applications of the same descent functions, in differing but irrelevant orders. (In fact, each minimal compressed descent DAG we have discussed is a compression of the minimal compressed descent DAG for commutative redundancy.) However, descent functions need not commute in order for an interpretation of $\mathscr{S}_n$ to define a redundant program.

Periodic redundancy, in which we assume the existence of integers i and j such that $c^i = d^j$ on $\mathscr{R}$ but do not assume that c and d commute, is explored in [6, sec. IV.F]. It is shown that each value in the descent tree of $f(x)$ can be expressed in a unique way in the form $e^k s(x)$, where $e = c^i = d^j$ and s is a string of descent function names that does not contain i consecutive c's or j consecutive d's. The shape of the minimal compressed descent DAG can be deduced from this fact, but we have not been able to formulate a transformation that eliminates the redundancy.

An interesting special case of periodic redundancy for which a transformation exists is a recursive program that returns the solution to the Tower of Hanoi

problem as a string. The program is as follows:

```
string procedure HANOI(n, from, to, using);
  value n, from, to, using; integer n; string from, to, using;
  if n = 0
    then HANOI := ''
    else HANOI := HANOI(n − 1, from, using, to)
                  || move(from, to)
                  || HANOI(n − 1, using, to, from)
```

(Recall that "||" represents string concatenation.) This is an instance of $\mathscr{S}$, with

$$x = \langle n, \mathit{from}, \mathit{to}, \mathit{using} \rangle,$$

$$p(x) = (x_1 = 0),$$

$$a(x) = \text{''}\text{''}$$

$$b(x, y, z) = y \parallel \mathit{move}(x_2, x_3) \parallel z,$$

$$c(x) = \langle x_1 - 1, x_2, x_4, x_3 \rangle,$$

and

$$d(x) = \langle x_1 - 1, x_4, x_3, x_2 \rangle.$$

(Since $c^2(x) = d^2(x) = \langle x_1 - 2, x_2, x_3, x_4 \rangle$ but c and d do not commute, *HANOI* exhibits periodic redundancy but not commutative periodic redundancy.) The value of n is fixed at each level of the descent tree, and there exist only six possible permutations of *from*, *to*, and *using*, so there can be no more than six distinct values at each level of the descent tree. In fact, there are at most three distinct values at each level of the tree, since three of the permutations can only result from an odd number of applications of c and d and the other three only from an even number. The descent tree and minimal compressed descent DAG for a typical computation of *HANOI* are shown in Figure 14.

Clearly, our taxonomy of redundancy is far from complete. In fact, redundancy exists whenever the number of distinct values obtainable by k or fewer applications of the descent functions of $\mathscr{S}_n$ is less than the number of nodes in a full n-ary tree of depth k, namely, $(n^{k+1} - 1)/(n - 1)$. The characterization of other forms of redundancy, and the formulation of program transformations applicable to those forms, are topics for further research. (See, for instance, [2].) The approach we have used here will, no doubt, be applicable in this research.

That approach consists of two steps. The first step is determination of the shape of the minimal compressed descent DAG. Given a descent condition consisting of a set of assertions of the form "$\alpha(x) = \beta(x)$ for all x in $\mathscr{R}$," where α and β are strings of descent function symbols, it is generally undecidable whether two given nodes of the descent tree are equivalent. Thus, it is not always possible to construct the minimal compressed descent DAG. There is, however, a large class of descent conditions for which the DAG is constructible. This class consists precisely of those descent conditions for which there is an effective mapping from a string of descent function names to a *canonical string*, such that all strings corresponding to equivalent values are mapped to the same string. We made use of canonical strings of the form c^i for explicit redundancy, g^i for common-generator redundancy, $e^k c^u d^v$ (with u and v in the appropriate range) for com-

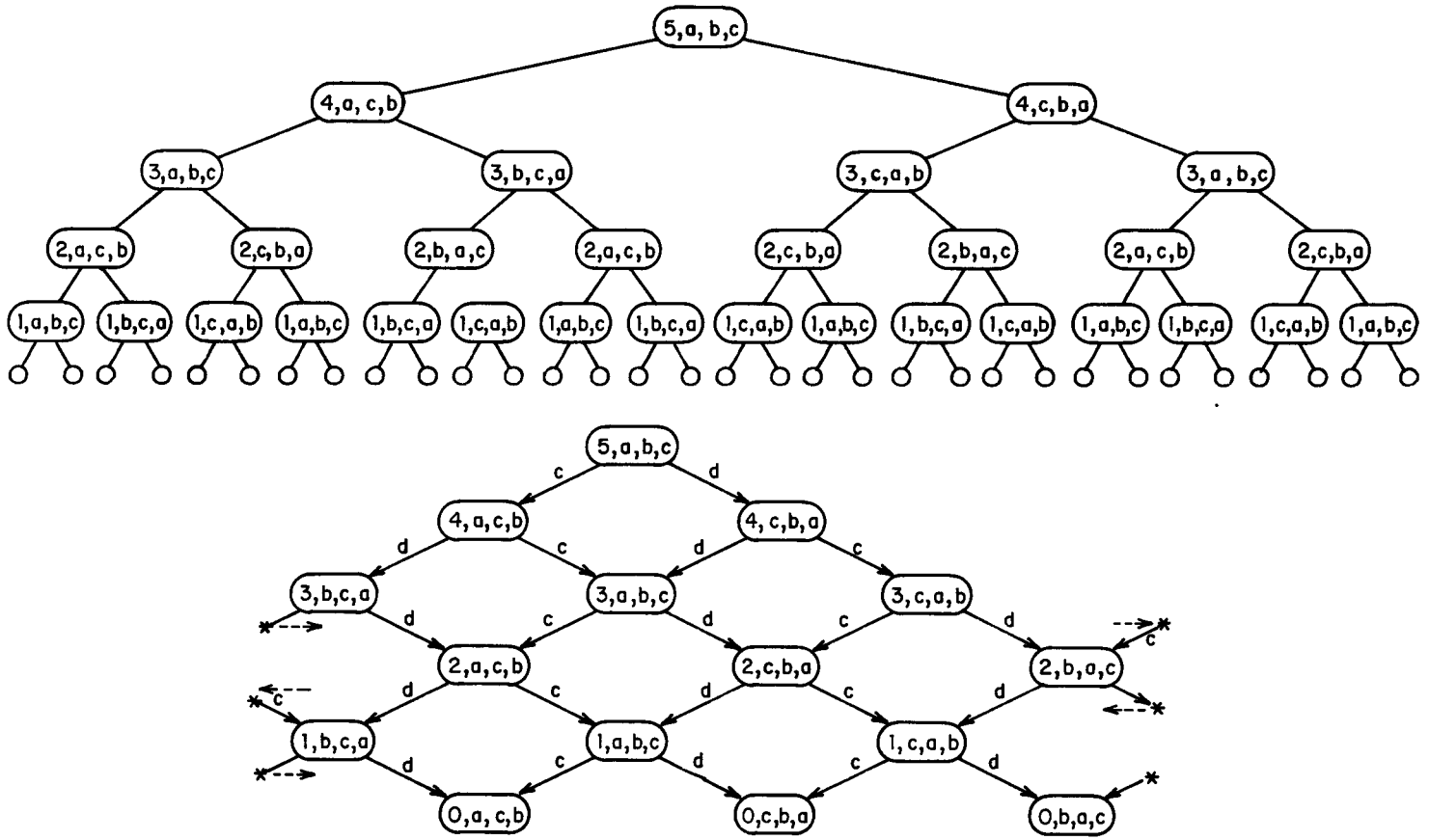


Fig. 14. The descent tree and minimal compressed descent DAG for the call *HANOI*(5, *a*, *b*, *c*). In the tree, left children represent applications of *c*, and right children represent applications of *d*. In the DAG, edges are labeled according to the descent functions they represent. Edges coming in from the right edge of the page are continuations of edges leaving the leftmost node of the previous level, and vice versa; the DAG does not get any wider after the third level.

mutative periodic redundancy, $c^i d^j$ for commutative redundancy, and $e^k s$, where there is a bound on the number of times each descent function name can occur consecutively in s, for periodic redundancy.

The second step is the search for a way of evaluating the expression represented by the compressed descent DAG without redundancy, using as little storage as possible. This is a "pebbling" problem. There is a considerable body of literature concerning the time–space trade-offs that may be achieved for the pebble game with various families of DAGs [15]. Since our goal is to compute results without redundancy, we want to pebble each node of a minimal compressed descent DAG only once. Thus, we are interested in achieving the lowest space bounds possible in an amount of time proportional to the number of nodes in the DAG.

ACKNOWLEDGMENTS

The author is grateful to Christos Papadimitriou, Thomas Cheatham, and especially Harry Lewis for their careful reading of the manuscript and many helpful suggestions. John Darlington, Peter Downey, and David Wise provided helpful references.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. BIRD, R.S. Tabulation techniques for recursive programs. *ACM Comput. Surv. 12*, 4 (Dec. 1980), 403–417.
3. BIRD, R.S. Improving programs by the introduction of recursion. *Commun. ACM 20*, 11 (Nov. 1977), 856–863.
4. BURSTALL, R.M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44–67.
5. CHANDRA, A.K. Efficient compilation of linear recursive programs. In Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory (Iowa City, Iowa, Oct. 1973), pp. 16–25.
6. COHEN, N.H. Source-to-Source Improvement of Recursive Programs. Ph.D. dissertation, Division of Applied Sciences, Harvard Univ., Cambridge, Mass., May 1980.
7. COHEN, N.H. Characterization and elimination of redundancy in recursive programs. In Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages (San Antonio, Tex., Jan. 29–31, 1979), pp. 143–157.
8. DARLINGTON, J. Program transformation and synthesis: Present capabilities. Res. Rep. 77/43, Dept. of Computing and Control, Imperial College of Science and Technology, London, Sept. 1977.
9. DARLINGTON, J., AND BURSTALL, R.M. A system which automatically improves programs. *Acta Inf. 6*, 1 (Mar. 1976), 41–60.
10. FRIEDMAN, D.P., WISE, D.S., AND WAND, M. Recursive programming through table look-up. In *SYMSAC '76*; proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation (Yorktown Heights, N.Y., Aug. 10–12, 1976), R.D. Jenks (Ed.), pp. 85–89.
11. HILDEN, J. Elimination of recursive calls using a small table of "randomly" selected function values. *BIT 16*, 1 (1976), 60–73.
12. LEWIS, H.R. A new decidable problem, with applications. In Proceedings, IEEE 18th Annual Symposium on Foundations of Computer Science (Providence, R.I., Oct.–Nov. 1977), pp. 62–73.
13. MICHIE, D. "Memo" functions and machine learning. *Nat. 218*, 5136 (Apr. 6, 1968), 19–22.
14. PATERSON, M.S., AND HEWITT, C.E. Comparative schematology. In Record of the Project MAC Conference on Concurrent Systems and Parallel Computation (Woods Hole, Mass., June 2–5, 1970), pp. 119–127.

15. PIPPENGER, N. Pebbling. Res. Rep. RC 8258, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., May 1980.
16. STRONG, H.R. Translating recursion equations into flowcharts. *J. Comput. Syst. Sci. 5*, 3 (June 1971), 254–285.
17. SWAMY, S., AND SAVAGE, J.E. Space-time tradeoffs for linear recursion. In Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages (San Antonio, Tex., Jan. 29–31, 1979), pp. 135–142.
18. VUILLEMIN, J. Correct and optimal implementations of recursion in a simple programming language. *J. Comput. Syst. Sci. 9*, 3 (Dec. 1974), 332–354.

Received May 1979; revised January and September 1982; accepted Sepember 1982