Information Processing Letters 17 (1983) 241–248
North-Holland 15 December 1983

AN APPLICATIVE RANDOM-ACCESS STACK *

Eugene W. MYERS

Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.

Communicated by G.R. Andrews
Received 1 June 1983
Revised 22 July 1983

Keywords: Applicative programming, linked list, number system, random-access stack

1. Introduction

Applicative (functional) programming has long been advocated on theoretical grounds as the formal properties of such programs are simple and elegant. Recently, there has been a trend to bring the applicative approach into the practical arena as a software development tool [2] and even as a programming vehicle [1,6]. Unfortunately, the requirement that operations be side-effect free makes efficient implementations difficult to achieve [6]. However, some researchers have begun to develop effective applicative implementations for classic data abstractions such as stacks, queues, tablets, lists, etc. [4,7].

In this article, applicative algorithms for a *random-access stack* (RA-stack) are presented. An RA-stack is a stack data abstraction in which one is further permitted to access the k*th* element of a stack. The approach uses a singly-linked list representation which is augmented with auxiliary pointers that permit the access of arbitrary stack elements in O(lg N) time were N is the size of the stack. The auxiliary linking structure is based on a number representation scheme for which subtraction by one involves at most one carry operation. Another scheme with this property appears in [3].

* This research was supported in part by the National Science Foundation under Grant MCS-8210096.

2. The skew-binary number system

A *skew-binary* number is a string consisting of the digits 0, 1 and 2. Each digit position represents a successive power of 2 *minus one.* That is the skew-binary number $\alpha = a_n a_{n-1} \cdots a_1$ denotes the integer value, $[\alpha] = \sum_{i=1}^{n} a_i(2^i - 1)$. For example, the skew-binary numbers 1000, 201 and 122 all denote the integer value $15_{(10)}$. Unlike common radix number systems, a given integer value can have more than one skew-binary representation.

A skew-binary number is *canonical* if all digits are 0 or 1 save for the lowest order non-zero digit, which may be 2. More formally, a string is a canonical skew-binary number if it is a member of the regular language

$$\mathrm{CSB} = (1(0+1)^* + \Lambda)(1+2)0^* + 0.$$

Lemma 1 demonstrates that the length of a CSB number is of the order of the base 2 logarithm of its value.

Lemma 1. *Suppose* $\alpha \in \mathrm{CSB}$, $|\alpha| = n$ *and* $[\alpha] = A$. *Then* $2^n - 1 \leqslant A \leqslant 2^{n+1} - 2$ *or equivalently* $n = \lfloor \lg(A+1) \rfloor$.

Proof. The CSB number of length n whose value is smallest is easily seen to be 10^{n-1} and its value is $2^n - 1$. The CSB number of largest value and length n must be of the form $1^j 2 0^k$ where $j + k + 1 = n$ as decrementing any digit in such a number decreases its value. But

$$\begin{aligned}[1^j 20^k] &= \sum_{i=k+2}^{n} (2^i - 1) + 2(2^{k+1} - 1)\\ &= 2^{n+1} - 1 - (n - k)\\ &\leqslant 2^{n+1} - 2 \quad \text{as } k \leqslant n - 1.\end{aligned}$$

This further implies that the number 20^{n-1} has the largest value. □

This critical feature of the CSB number system is that every integer value has a unique CSB representation. This is proved in Lemma 2.

Lemma 2. *Each integer value has a unique* CSB *representation.*

Proof. It is first shown that each integer value is represented by some CSB number. To do so it suffices to show by induction on $n > 0$ that if $A \leqslant 2^{n+1} - 2$, then there exists an α such that $[\alpha] = A$. The basis of the induction is easy:

$$n = 1 \Rightarrow A \leqslant 2 \Rightarrow A \in \{[0],[1],[2]\}.$$

Suppose the induction hypothesis is true for $n = k$ and that $A \leqslant 2^{k+2} - 2$. If $A \leqslant 2^{k+1} - 2$, then an α such that $[\alpha] = A$ exists by the induction hypothesis. If $A = 2^{k+2} - 2$, then $A = [20^k]$. The case where $2^{k+1} - 1 \leqslant A \leqslant 2^{k+2} - 3$ remains. The induction hypothesis assures that there exists a β such that $[\beta] = A - 2^{k+1} + 1 \leqslant 2^{k+1} - 2$ and Lemma 1 asserts that $|\beta| \leqslant k$. Thus $[10^{k-|\beta|}\beta] = 2^{k+1} - 1 + [\beta] = A$.

It remains to demonstrate that each integer value is represented by a unique CSB number. It suffices to show that if $\alpha \neq \beta$, then $[\alpha] \neq [\beta]$. If $\alpha \neq \beta$, then without loss of generality either $|\alpha| \geqslant |\beta| + 1$ or there exists a ρ such that

$$\alpha = \rho a \tau, \quad \beta = \rho b \pi, \quad |\tau| = |\pi| \quad \text{and} \quad a \geqslant b + 1.$$

If $|\alpha| \geqslant |\beta| + 1$, then, by Lemma 1,

$$[\alpha] \geqslant 2^{|\alpha|} - 1 \geqslant 2^{|\beta|+1} - 1 > 2^{|\beta|+1} - 2 \geqslant [\beta].$$

In the other case assume $n = |\tau|\ (= |\pi|)$. By Lemma 1,

$$\begin{aligned}[\alpha 0^n] &\geqslant [(b + 1)0^n] = [b0^n] + (2^{n+1} - 1)\\ &> [b0^n] + [\pi] = [b\pi]\end{aligned}$$

and thus

$$\begin{aligned}[\alpha] &= [\rho a \tau] \geqslant [\rho 0^{n+1}] + [\alpha 0^n]\\ &> [\rho 0^{n+1}] + [b\pi] = [\rho b \pi] = [\beta]. \qquad □\end{aligned}$$

Suppose $\alpha > 2$ is a CSB number and that $\alpha = \rho a 0^k$ where a is the lowest order non-zero digit and $k > 0$. Then $\beta = \rho(a - 1)20^{k-1}$ is the predecessor of α as $a \in \{1, 2\}$ implies $\beta \in$ CSB and

$$\begin{aligned}[\rho a 0^k] - 1 &= [\rho(a - 1)0^k] + (2^{k+1} - 1) - 1\\ &= [\rho(a - 1)0^k] + 2[2^k - 1]\\ &= [\rho(a - 1)20^{k-1}].\end{aligned}$$

Thus a CSB number and its predecessor differ only in the lowest order non-zero digit and the position to its immediate right (if it exists). Alternately, in the CSB number system subtraction by one involves at most one carry operation.

3. An applicative random-access stack

In many applications it is desirable to view a stack data structure as but one instance of an object from a 'stack' data abstraction. For example, programming languages can be viewed as linguistic vehicles for manipulating data abstractions such as 'integer', 'array', 'string', etc. Formally, we will assume that a *value-semantic stack data abstraction* consists of the following:

(a) An arbitrary and time varying number of *objects* of type stack with homogeneous domain X. (Homogeneity is not necessary but is assumed for simplicity.)

(b) A fixed finite collection of *operators* that access and manipulate objects.

(c) A time varying set of *variables*, each of which *refers* to an object. Variables can be created and destroyed, and their reference relation can be modified by *assignment*. A variable denotes the *value* of the object to which it refers. Only assignment can change the value of a variable.

The *state* of the data abstraction is the collection of objects referred to by the current set of variables. A generally accepted stack operation repertoire [5] consists of the following four primitives:

(1) EMPTY(S) : Boolean	Determine if stack S contains any elements.
(2) TOP(S) : X	Return the top (last) element of stack S.
(3) POP(S) : Stack_of_X	Delete the top element of stack S.
(4) PUSH(S, x) : Stack_of_X	Append x as the top element of stack S.

Observe that the primitives have been formulated as functions. The value-semantic constraint that only an assignment can change the value of a variable forces an implementation in which an operator may not modify the value(s) contained in its operand object(s), i.e., it must operate *applicatively*. We call such implementations *applicative data types*.

The elementary singly-linked list model of a stack [5] is naturally suited to the formulation of an applicative data type. Each variable is modeled as a pointer to the head of a list. PUSH operations are immediately applicative as the process of adding a cell to the end of a list has no side-effect upon the list. POP operations are made applicative by returning a pointer to the element following the head of the list *without* deleting the head of the list. (However, this cell is presumably garbage collected if it is no longer a relevant part of the current state of the abstraction; i.e., no variable references an object (list) of which the cell is a component.) [1] All operations require O(1) time and PUSH operations require O(1) space. Fig. 1 depicts a snapshot of the state of a stack data abstraction and illustrates the effect of the operations. Observe that each list is terminated by a unique 'Nil' cell.

We momentarily digress to convince the reader of the difficulty in extending alternative stack models to an applicative context. For example, another elementary model consists of an array for storing the stack elements in sequence and an

[1] The issue of how the list space of the abstraction is garbage collected is not treated here. A reference counter scheme attributable to Weizenbaum [8] suffices and proves O(1) on-line allocation and collection primitives.

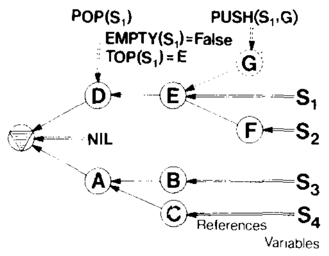

Fig. 1. Sample state of an applicative stack data type.

integer variable giving the index of the top element [5]. In order to be applicative, the PUSH operation must not have a side-effect upon the array. This can be done by taking linear time and space to make a copy of the entire array. This prohibitive cost can be reduced to logarithmic time and space using the method in [7]. While an O(1) time and space method cannot be ruled out, it is certain that it would not be as simple and natural as the linked-list approach.

A *random-access stack* (RA-stack) abstraction is an extension of the stack abstraction in which the EMPTY and TOP operators are superceded by the following operators:

(1′) LENGTH(S) : Integer	Return the number of elements in stack S.
(2′) FIND(S, k) : X	Return the k*th* (from the bottom) element of stack S.

In this abstraction one may query the length of an RA-stack and access an arbitrary element by index. By additionally retaining in each cell its distance from the terminating 'Nil' cell, the linked list model sketched above suffices to realize an applicative RA-stack data type. The shortcoming of this solution is in the complexity of the FIND operation which requires a linear sequential search for the k*th* element.

The complexity of FIND can be reduced to O(lg N) time (N is the length of the stack in question) without increasing the complexity of the other operators by augmenting each cell in the linked list model with an auxiliary pointer based on the skew-binary number system. Suppose each cell has the following record structure:

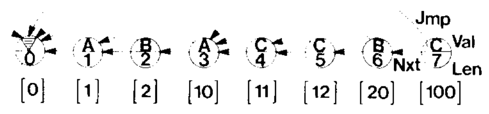

Fig. 2. The structure of an RA-stack.

```
Type cell = Record
                NXT, JMP : ↑cell
                      LEN : integer
                      VAL : X
             End
Type Stack = ↑cell
```

The NXT-field contains the standard list pointer, the LEN-field contains the distance of the cell from its list's origin, and the VAL-field contains an element of type X. The JMP-field contains the auxiliary pointer needed to reduce search times and its value is formulated as follows. Let J be the function that maps a positive integer A into the value of the CSB number obtained by subtracting one from the lowest non-zero digit in the CSB representation of A. Formally, $J(A) = [\rho(a-1)0^n]$ if there exist ρ, $a > 0$, and n such that $A = [\rho a 0^n]$ and $\rho \neq \epsilon$ or $a = 2$; $J(A) = 0$ otherwise. Note that the convention $J(0) = 0$ is adopted. For a cell S (including the 'Nil' cell), S.JMP contains a pointer to the cell T for which T.LEN = J(S.LEN).[2] Fig. 2 illustrates the RA-stack list model; the JMP-pointers are indicated by dashed arrows.

The algorithms for the RA-stack operations are now presented. The functions for LENGTH and POP are straightforward:

```
Function LENGTH(s : Stack) : Integer
1.  LENGTH ← s↑.LEN
```

```
Function POP(s : Stack) : Stack
(* Assume the pre-condition: LENGTH(s) ≠ 0 *)
1.  POP ← s↑.NXT.
```

[2] One may immediately question why a JMP-pointer scheme based on a conventional radix number system is not used. The answer is that the analogous algorithms would not be as efficient. PUSH would require $O(\lg N)$ time and FIND would require $O(\lg^2 N)$ time. Intuitively, the difference is that subtraction by one requires $O(\lg N)$ carries in a radix-number system as opposed $O(1)$ carries in the CSB number system. The difference can be more formally perceived through the careful study of Lemmas 3 and 4.

The FIND algorithm searches for the k*th* element by using a JMP-pointer whenever its use does not take one past the k*th* element. Otherwise, the NXT-pointer is followed.

```
Function FIND(s : Stack; k : Integer) : X
(* Assume the pre-condition:
 k ∈ [1, LENGTH(s)] *)
1.  While s↑.LEN ≠ k Do
2.     If s↑.JMP↑.LEN < k Then
3.        s ← s↑.NXT
4.     Else
5.        s ← s↑.JMP
6.  FIND ← s↑.VAL
```

The PUSH algorithm appends a cell to the linked list in lines 6. through 9. The determination of the JMP-pointer for this new cell is embodied in lines 1. through 5. The validity of this code fragment is deferred to Lemma 3 below.

```
Function PUSH(s : Stack; x : X) : Stack;
  Var p, t : ↑cell;
 1.  t ← s↑.JMP
 2.  If s↑.LEN − t↑.LEN = t↑.LEN
         − t↑.JMP↑.LEN Then
 3.     t ← t↑.JMP
 4.  Else
 5.     t ← s
 6.  New(p)
 7.  p↑.NXT ← s
 8.  p↑.JMP ← t
 9.  p↑.VAL ← x
10.  p↑.LEN ← s↑.LEN + 1
11.  PUSH ← p
```

The correctness of the algorithms for LENGTH and POP is immediate. The (partial) correctness of FIND with respect to the precondition $1 \leqslant k \leqslant$ LENGTH(s) follows from the invariant s↑.LEN $\geqslant$ k for the search loop in lines 1. through 5. FIND must terminate as each loop iteration strictly reduces the value of s↑.LEN. The only difficulty in showing PUSH is correct is in verifying that the JMP-pointer is correctly determined in lines 1. through 5. This follows from the property of the function J stated and proved in Lemma 3.

Lemma 3. *Let* EQJ *be the predicate*

$$A - J(A) = J(A) - J(J(A)).$$

If EQJ *is true, then* J(A + 1) = J(J(A)); *otherwise* J(A + 1) = A.

Proof. Suppose $A = [\alpha]$ where $\alpha \in$ CSB and let ONES = 1(0 + 1)*. In the next paragraph it is shown that if $\alpha \in$ ONES, then J(A + 1) = A and EQJ does not hold. In the last paragraph it is shown that if $\alpha \notin$ ONES, then J(A + 1) = J(J(A)) and EQJ does hold. The lemma then follows as, from these facts, one can infer that

$$\text{EQJ} \Rightarrow \alpha \notin \text{ONES} \Rightarrow J(A+1) = J(J(A))$$

and not

$$\text{EQJ} \Rightarrow \alpha \in \text{ONES} \Rightarrow J(A+1) = A.$$

If $\alpha \in$ ONES, then either α contains one '1' or α contains two or more '1's, i.e., $\alpha = 10^k$ for $k \geqslant 0$ or $\alpha = \rho 10^j 10^k$ for $j, k \geqslant 0$. But

$$A = [10^k] \Rightarrow J(A) = 0$$

and

$$J(J(A)) = 0 \Rightarrow A - J(A) = 2^{k+1} - 1 \neq 0$$
$$= J(A) - (J(A)).$$

Also

$$A = [\rho 10^j 10^k] \Rightarrow J(A) = [\rho 10^{j+(k+1)}]$$

and

$$J(J(A)) = [\rho 0^{(j+1)+(k+1)}] \Rightarrow$$
$$\Rightarrow A - J(A) = 2^{(k+1)} - 1 \neq 2^{(j+1)+(k+1)} - 1$$
$$= J(A) - J(J(A)).$$

Thus in either case EQJ does not hold. If $\alpha \in$ ONES, then either $\alpha = \rho 10^k$ for $k > 0$ or $\alpha = \rho 1$. But

$$A = [\rho 10^k] \Rightarrow A + 1 = [\rho 10^{k-1} 1]$$
$$\Rightarrow J(A+1) = [\rho 10^k] = A$$

and

$$A = [\rho 1] \Rightarrow A + 1 = [\rho 2]$$
$$\Rightarrow J(A+1) = [\rho 1] = A.$$

Thus in either case J(A + 1) = A.

If $\alpha \notin$ ONES, then either α's non-zero digit of lowest order is 2 or α is 0, i.e., $\alpha = \rho 20^k$ for $k \geqslant 0$ or $\alpha = 0$. But

$$A = [\rho 20^k] \Rightarrow J(A) = [\rho 10^k]$$

and

$$J(J(A)) = [\rho 0^{k+1}] \Rightarrow A - J(A) = 2^{k+1} - 1$$
$$= J(A) - J(J(A)).$$

Also,

$$A = 0 \Rightarrow J(A) = J(J(A)) = 0$$
$$\Rightarrow A - J(A) = 0 = J(A) - J(J(A)).$$

Thus in either case EQJ does hold. Moreover,

$$A = [\rho a 20^k] \Rightarrow A + 1 = [\rho(a+1)0^{k+1}]$$
$$\Rightarrow J(A+1) = [\rho a 0^{k+1}]$$
$$= J(J(A));$$

$$A = [20^k] \Rightarrow A + 1 = [10^{k+1}]$$
$$\Rightarrow J(A+1) = 0 = J(J(A))$$

and

$$A = 0 \Rightarrow A + 1 = 1 \Rightarrow J(A+1) = 0 = J(J(A)).$$

In all cases, J(A + 1) = J(J(A)). □

4. Performance analysis

Each of the operations—PUSH, POP and LENGTH—require O(1) time as each is a straight-line algorithm. [3] PUSH is the only operation requiring space. In consumes O(1) space as one cell is allocated per call. It is claimed that FIND requires O(lg N) time where N is the size of the stack in question. This comparatively coarse claim will follow from the detailed analysis of FIND given in the remainder of this section.

Lemma 4 gives an exact formula for the number of times the search loop of FIND(S, k) is re-

[3] A complete analysis requires that the cost of managing storage be examined. The scheme of Weizenbaum [8] mentioned in Footnote 1 permits the *inclusion* of these costs in the O(1) complexity claims.

peated in terms of the digits of the CSB representations of the length of S and the integer k. From an intuitive point of view, the lemma gives the number of applications of the function J and 'subtraction-by-one' needed to transform a CSB string α into a smaller non-zero CSB string β.

Lemma 4. *Suppose* LENGTH(S) $= [\alpha]$, $k = [\beta]$ *and* $0 < k <$ LENGTH(S). *Further suppose that* $\alpha = a_n a_{n-1} \cdots a_1$ *and* $0^{n-|\beta|}\beta = b_n b_{n-1} \cdots b_1$. *The search loop of* FIND(S, k) *is repeated exactly*

$$\sum_{i=1}^{U-1} a_i + (a_U - b_U) + \sum_{i=L}^{U-1} (2 - b_i) \qquad (1)$$

times, where

$$U = \max_i \{a_i \neq b_i\} \quad \textit{and} \quad L = \min_i \{b_i \neq 0\}.$$

Proof. First note that $[\beta] \neq 0$ implies L exists and $[\alpha] \neq [\beta]$ implies U exists. In what follows lines 1. through 5. of FIND are referred to as the search loop, line 3. is referred to as a next step, and line 5. is referred to as a jump step. In addition, notation s_i denotes the value of the variable s after i iterations of the search loop and notation J^i denotes the *ith* power of function J. A statement of the form $s_i \uparrow.\text{LEN} = [\tau]$ is assumed to assert not only that the equality holds, but also that the search loop of FIND is repeated at least i times. It is further assumed that ρ is the common prefix of α and β, i.e., $\rho = a_n \cdots a_{U+1} = b_n \cdots b_{U+1}$.

Let $A(m) = \sum_{i=1}^{m} a_i$. It is shown by induction on m that, for all $m < U$,

$$s_{A(m)} \uparrow.\text{LEN} = [\rho a_U \cdots a_{m+1} 0^m].$$

The basis follows as at the start of the algorithm:

$$s_{A(0)} \uparrow.\text{LEN} = s_0 \uparrow.\text{LEN} = [\alpha] = [\rho a_U \cdots a_1].$$

Now assume $k < U$ and the induction hypothesis for $m = k - 1$. Then,

$$\begin{aligned} J^{a_k}(s_{A(k-1)} \uparrow.\text{LEN}) &= J^{a_k}([\rho a_U \cdots a_k 0^{k-1}]) \\ &= p[\rho a_U \cdots a_{k+1} 0^k] > [\beta] \end{aligned}$$

as the converse would imply $[\alpha] \leqslant [\beta]$ (see Lemma 2). Thus after $A(k-1)$ iterations, the search loop will continue with a_k jump steps and $s_{A(k)} \uparrow.\text{LEN}$ $= [\rho a_U \cdots a_{k+1} 0^k]$.

Let

$$B(m) = A(U-1) + (a_U - b_U) + \sum_{i=m}^{U-1} (2 - b_i).$$

It is shown by induction on decreasing values of m that if $U \geqslant m > L$, then

$$s_{B(m)} \uparrow.\text{LEN} = [\rho b_U \cdots b_m 2 0^{m-2}].$$

For the basis $m = U$, observe that $B(U) = A(U-1) + (a_U - b_U)$ and, from the paragraph above, $s_{A(U-1)} \uparrow.\text{LEN} = [\rho a_U 0^{U-1}]$. Then

$$J^{(a_U - b_U) - 1}([\rho a_U 0^{U-1}]) - 1 = [\rho b_U 2 0^{U-2}] \geqslant [\beta]$$

as 20^{U-2} is the largest CSB number of $U - 1$ digits. Moreover,

$$J^{a_U - b_U}([\rho a_U 0^{U-1}]) = [\rho b_U 0^{U-1}] < [\beta]$$

as $L < U$ implies $[b_{U-1} \cdots b_1] > 0$. Thus after $A(U-1)$ iterations, the search loop will continue with $(a_U - b_U) - 1$ jump steps followed by a single next step and $s_{B(U)} \uparrow.\text{LEN} = [\rho b_U 2 0^{U-2}]$. Now assume $k > L$ and the induction hypothesis for $m = k + 1$, i.e.,

$$s_{B(k+1)} \uparrow.\text{LEN} = [\rho b_U \cdots b_{k+1} 2 0^{k-1}].$$

(Note that $k > L$ implies $b_k < 2$.) Then,

$$\begin{aligned} &J^{1-b_k}([\rho b_U \cdots b_{k+1} 2 0^{k-1}]) - 1 = \\ &\quad = [\rho b_U \cdots b_k 2 0^{k-2}] \geqslant [\beta] \end{aligned}$$

as 20^{k-2} is the largest CSB number of $k - 1$ digits. Moreover,

$$\begin{aligned} &J^{2-b_k}([\rho b_U \cdots b_{k+1} 2 0^{k-1}]) = \\ &\quad = [\rho b_U \cdots b_k 0^{k-1}] < [\beta] \end{aligned}$$

as $L < k$ implies $[b_{k-1} \cdots b_1] > 0$. Thus, after $B(k+1)$ iterations, the search loop will continue with $1 - b_k$ jump steps followed by a single next step and $s_{B(k)} \uparrow.\text{LEN} = [\rho b_U \cdots b_k 2 0^{k-2}]$.

Finally, it is shown that $s_{B(L)} \uparrow.\text{LEN} = [\beta]$. First consider the case in which $L \geqslant U$. Then, $B(L) = B(U)$ and, from the last paragraph,

$$J^{a_U - b_U}(s_{A(U-1)} \uparrow.\text{LEN}) = [\rho b_U 0^{U-1}]$$

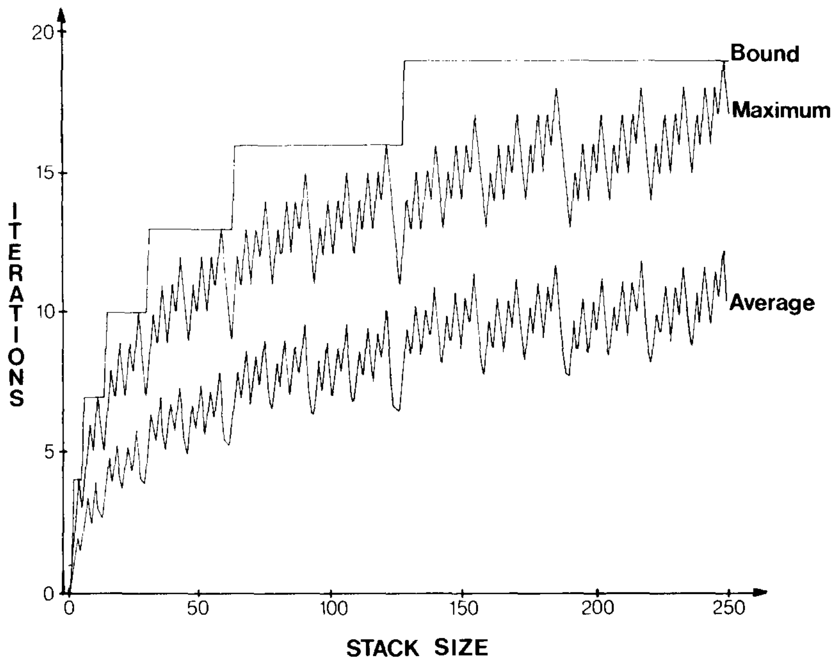

Fig. 3. The worst and average performance of FIND.

which equals $[\beta]$ as $L \geqslant U$ implies $[b_{U-1} \cdots b_1] = 0$. Thus, after $A(U-1)$ iterations, the search loop takes $a_U - b_U$ final jump steps and $s_{B(L)} \uparrow .LEN = [\beta]$. When $L < U$, it follows from the last paragraph that

$$J^{2-b_L}(s_{B(L+1)} \uparrow .LEN) = [\rho b_U \cdots b_L 0^{L-1}]$$

which equals $[\beta]$ as $[b_{L-1} \cdots b_1] = 0$ by the definition of L. Thus, after $B(L+1)$ iterations, the search loop takes $2 - b_L$ final jump steps and $S_{B(L)} \uparrow .LEN = [\beta]$.

The proof concludes with the observation that $B(L)$ is given by (1) and thus $s_{(1)} \uparrow .LEN = [\beta]$. □

The next result uses Lemma 4 to place an upper bound on the number of iterations of the search loop in terms of the size of stack S.

Corollary 5. *Suppose* $N = \text{LENGTH}(S)$. *Then, for all* k, *the search loop of* FIND(S, k) *is repeated at most* $3\lfloor \lg(N+1) \rfloor - 2$ *times.*

Proof. Suppose $k = [\beta]$ and $\text{LENGTH}(S) = [\alpha]$ where the length of α is n. Consider formula (1) given in Lemma 4. For a given n, the sub-expression $\sum_{i=1}^{U-1} a_i + (a_U - b_U)$ attains a maximum value of $n+1$ when $U = n$, $b_U = 0$ and $\alpha = 1^{n-1}2$ (the CSB number of length n whose digit sum is maximal). The sub-expression $\sum_{i=L}^{U-1}(2 - b_i)$ attains a maximum value of $2n-3$ when $U = n$, $L = 1$ and $\beta = 1$. Both maximums can be met simultaneously when $\alpha = 1^{n-1}2$ and $\beta = 1$ in which case (1) evaluates to $3n-2$. The result then follows as Lemma 1 implies $n = \lfloor \lg(N+1) \rfloor$. □

Observe from the proof that the bound of Corollary 5 is tight in that it is exactly met by FIND(S, 1) whenever $\text{LENGTH}(S) = 2^k - k$ for some $k > 1$. The plot of Fig. 3 shows the upper bound, worst-case number of iterations and average-case number of iterations of FIND as a function of stack size. A uniform distribution of the index k was assumed for the average-case plot.

Acknowledgment

The author is indebted to Gary Levin for his many astute suggestions. The author also wishes to thank the referees for their helpful comments.

References

[1] J. Bachus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Comm. ACM 21 (8) (1978) 613-641.
[2] M. Broy and P. Pepper, Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite algorithm, ACM Trans. Programming Languages and Systems 4 (3) (1982) 362-381.
[3] M. Furer, The tight deterministic time hierarchy, Proc. 14th ACM Symp. on Theory of Computing (1982) pp. 8-16.
[4] R. Hood and R. Melville, Real-time queue operations in pure LISP, Inform. Process. Lett. 13 (2) (1981) 50-54.
[5] E. Horowitz and S. Sahni, Fundamentals of Data Structures (Computer Science Press, Potomac, MD, 1976) pp. 79-81, 112-114.
[6] J.H. Morris, E. Schmidt and P. Wadler, Experience with an applicative string processing language, Proc. 7th ACM Symp. on the Principles of Programming Languages (1980) pp. 32-46.
[7] E. Myers, AVL-Dags: An applicative list model, Tech. Rept. TR82-9, Dept. of Computer Science, Univ. of Arizona, Tucson, AZ, 1982.
[8] J. Weizenbaum, Symmetric list processor, Comm. ACM 6 (9) (1963) 524-536.