Iterative Procedures for Computing Ackerman's Function

M. P. Ward*
Martin.Ward@durham.ac.uk
http://www.dur.ac.uk/~dcs0mpw/

July 16, 1993

Contents

1 Introduction 2

2 Notation 2

3 The Transformations 3

3.1 Back Expansion of a Conditional . . . 3

3.2 Forward Expansion . . . 3

3.3 Double Iteration . . . 4

3.4 Proper Inversion . . . 4

4 The Function 4

5 Method A: The "direct method" 5

5.1 Alternative Application of Method A . . . 7

6 Method B: The "postponed obligations" method 7

6.1 Alternative application of method B . . . 9

7 Method C 10

8 Direct Proof of Termination 13

8.1 Determining the number of steps . . . 15

9 Conclusion 22

10 References 22

Abstract

This paper uses Ackerman's function as a testbed to illustrate the operation of various program transformations which take recursive procedures to equivalent iterative forms. The transformations are taken from the author's DPhil thesis [19]. In this paper we illustrate that

*Department of Computer Science University of Durham, Durham, UK

they can be successfully applied to even the most convoluted recursion. For many programs a recursive function is the most natural and clear specification while an iterative (or tail-recursive) form is the most efficient implementation. This paper illustrates how an efficient iterative program can be developed and verified by starting with a simple recursive program and using proven transformations to remove the recursion. The resulting iterative program will be correct by construction, so the problem of a direct verification of the iterative algorithm is avoided. This process can also throw light on the nature of the recursive specification. Several interesting properties of Ackermann's function and the iterative algorithms are derived in the course of this development.

1 Introduction

Ackerman's function is defined for all $m, n \geqslant 0$ as follows:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m,0) &= A(m-1,1) \quad \text{for } m > 0 \\ A(m,n) &= A(m-1,\mathbf{A}(m,n-1)) \quad \text{for } m,n > 0. \end{aligned}$$

Ackerman's function was originally constructed in order to prove that there exist computable functions which cannot be defined using primitive recursion. The set of primitive recursive functions is constructed from a small set of "elementary" functions (the constant functions and the successor function "add one") extended by the use of simple recursion. Simple recursion means:

If c is a constant and G and h are primitive recursive functions, then the function f defined:

$$\begin{aligned} f(0) &= c \\ f(n+1) &= G(h(n+1), f(n)) \end{aligned}$$

is also primitive recursive.

So addition can be defined using recursion on the successor function, multiplication by repeated addition, and so on.

Most of the functions and procedures traditionally used to illustrate recursion removal are simple recursions; (the "Towers of Hanoi" problem is a common example, as is the factorial function). Ackermann proved [2] that his function could not be defined from the successor function using primitive recursion alone. This suggests that its form of recursion is more "complicated" than primitive recursions and therefore should present a stronger test of the techniques of recursion removal.

2 Notation

$\mathbf{S}$, $\mathbf{S}_1$, $\mathbf{S}_2$ etc. are statements. Δ is a set of formulae of first order logic which includes all the "assumptions" we are making about the functions and relations used in programs. For example, if we are using the symbol "+" to represent addition then Δ will include the properties of addition, written in terms of "+", for example $(\forall x, y.\, x + y = y + x)$, $(\forall x.\, x + 0 = x)$ and so on. Note that the formulae must have all their variables quantified.

Following the notation of [4] and [19]:

$\Delta \vdash \mathbf{S}_1 \leqslant \mathbf{S}_2$ means that statement $\mathbf{S}_1$ is refined by statement $\mathbf{S}_2$ under the assumption that all the formulae in Δ are true. This means: For each initial state on which $\mathbf{S}_1$ is guaranteed to terminate, $\mathbf{S}_2$ is also guaranteed to terminate and will terminate in one of the allowed final states of $\mathbf{S}_1$.

$\Delta \vdash \mathbf{S}_1 \approx \mathbf{S}_2$ means $\Delta \vdash \mathbf{S}_1 \leqslant \mathbf{S}_2$ and $\Delta \vdash \mathbf{S}_2 \leqslant \mathbf{S}_1$.

If this is the case then wherever we see $\mathbf{S}_1$ in a program we may replace it by $\mathbf{S}_2$ without changing the input/output behaviour of the program. (See [19] for the proof of this assertion). In this case we say $\mathbf{S}_1$ and $\mathbf{S}_2$ are *equivalent*.

We use the following notation for sequences:

- If L is a sequence then we denote the ith element of L by $L[i]$.
- If L has the n elements $a_1, a_2, \ldots, a_n$ then we denote the whole sequence by $\langle a_n, \ldots, a_2, a_1 \rangle$.
- If L has n elements then $\ell(L) = n$.
- If k is an integer between 1 and n then the sequence formed from the kth element onwards is defined as:

$$L[k\,..] = \langle L[n], \ldots, L[k] \rangle$$

- The sequence formed from the first k elements, called the *restriction* of L to k is defined as:

$$L \restriction k = \langle L[k], \ldots, L[1] \rangle$$

- If L_1 and L_2 are sequences then the *concatenation* of L_1 and L_2 is defined as:

$$L_1 + L_{22} = \langle L_1[\ell(L_1)], \ldots, L_1[1], L_2[\ell(L_2)], \ldots, L_2[1] \rangle$$

We have two special statements which are used to implement stacks as sequences. The statements add and remove a single element from the given stack: If L is a sequence and e an expression then $L \xleftarrow{\text{push}} e$ is equivalent to the assignment: $L := L + \langle e \rangle$. If L is a sequence and x a variable then $x \xleftarrow{\text{pop}} L$ is equivalent to $x := L[1];\ L := L[2\,..]$.

So for example, if x is a variable and L a sequence then: $L \xleftarrow{\text{push}} x;\ x \xleftarrow{\text{pop}} L$ leaves both x and L unchanged.

3 The Transformations

We will use the following basic transformations in several of the derivations, (the formal proofs of these are given in [19]):

3.1 Back Expansion of a Conditional

If formula $\mathbf{B}$ is invariant over statement $\mathbf{S}$ then:

$$\Delta \vdash \mathbf{S};\ \textsf{if}\ \mathbf{B}\ \textsf{then}\ \mathbf{S}_1\ \textsf{else}\ \mathbf{S}_2\ \textsf{fi} \ \approx\ \textsf{if}\ \mathbf{B}\ \textsf{then}\ \mathbf{S};\ \mathbf{S}_1\ \textsf{else}\ \mathbf{S};\ \mathbf{S}_2\ \textsf{fi}$$

3.2 Forward Expansion

$$\Delta \vdash \textsf{if}\ \mathbf{B}\ \textsf{then}\ \mathbf{S}_1\ \textsf{else}\ \mathbf{S}_2\ \textsf{fi};\ \mathbf{S} \ \approx\ \textsf{if}\ \mathbf{B}\ \textsf{then}\ \mathbf{S}_1;\ \mathbf{S}\ \textsf{else}\ \mathbf{S}_2;\ \mathbf{S}\ \textsf{fi}$$

(These are also described as "moving a statement into an **if** statement")

Our programming language includes loops of the form **do S od** which are terminated by the execution of a statement of the form **exit**(n) (where n is an integer) within the loop. This statement causes immediate termination of n enclosing **do** loops. A sub-statement in a statement which if executed would cause termination of the statement is called a terminal statement, its terminal value is the number of enclosing **do** loops which would also be terminated by that statement. For example in:

$$\textsf{if}\ y = 1\ \textsf{then}\ x := 1\ \textsf{else}\ x := 2\ \textsf{fi}$$

the terminal statements are the two assignments to x, they both have terminal value zero. In:

$$\textbf{do if } y = 1 \textbf{ then } x := 1; \textbf{ exit else } x := 2; \textbf{ exit}(2) \textbf{ fi od}$$

The terminal statements are **exit** (with terminal value zero) and **exit**(2) (with terminal value one).

This form of loop is discussed in [9], [17] and other places. Buhr [10] suggests that it is the most "natural" representation for many loops.

Definition 3.1 **S** *is* reducible *if replacing any terminal statement* **exit**(k)*, which has terminal value one, by* **exit**$(k-1)$ *gives a terminal statement of* **S**.

Thus the last example given is a reducible statement.

3.3 Double Iteration

If **S** is reducible then

$$\Delta \vdash \textbf{do do S od od} \approx \textbf{do S} - 1 \textbf{ od}$$

where $\mathbf{S} - 1$ is **S** with all the **exit** statements reduced by one (Thus **exit**(2) becomes **exit** and **exit** becomes a **skip** statement ie a statement which has no effect).

3.4 Proper Inversion

If every terminal statement of $\mathbf{S}_1$ has terminal value zero then

$$\Delta \vdash \textbf{do } \mathbf{S}_1;\ \mathbf{S}_2 \textbf{ od} \approx \mathbf{S}_1; \textbf{ do } \mathbf{S}_2;\ \mathbf{S}_1 \textbf{ od}$$

Program transformations are used in program development in [5], [6], [12], [13] and [7]. These workers start with an applicative kernel language and add procedural constructs by means of "definitional transformations". Their methods cannot cope with general specifications and become cumbersome and unwieldy as more constructs (such as loops with multiple **exit**s and expressions with side-effects) are added. In contrast, the theory developed in [19] uses a procedural kernel language which is more easily extended and is able to accommodate specifications written in terms of first order logic.

4 The Function

A recursive function to compute Ackerman's function is:

funct $A(m, n) \equiv$
  if $m = 0$ **then** $n + 1$
  elsif $n = 0$ **then** $A(m - 1, 1)$
      else $A(m - 1, A(m, n - 1))$ **fi.**

It is easy to see that this terminates since the pair $\langle m, n\rangle$ is decreased under the lexical order denoted $\prec$. The function clearly terminates for $n = m = 0$ so consider $\langle m, n\rangle \neq \langle 0, 0\rangle$ such that **A** terminates for all lower pairs in the lexical order. If $m = 0$ the result is trivial. If $n = 0$ then $\langle m - 1, n\rangle \prec \langle m, n\rangle$ so $A(m - 1, n)$ terminates whence $A(m, n)$ terminates. If $n, m > 0$ then $\langle m, n - 1\rangle \prec \langle m, n\rangle$ so $A(m, n - 1)$ terminates, hence $\langle m - 1, A(m, n - 1)\rangle \prec \langle m, n\rangle$ so $A(m - 1, A(m, n - 1))$ terminates and the result is proved. This proof can be made rigorous by applying the theorem on recursive implementation of specifications in [19].

This function is equivalent to the following procedure (which sets r to $A(m, n)$):

proc $\mathbf{A}_p(m, n) \equiv$
  if $m = 0$ **then** $r := n + 1$
  elsif $n = 0$ **then** $\mathbf{A}_p(m - 1, 1)$
      else $\mathbf{A}_p(m, n - 1)$; $\mathbf{A}_p(m - 1, r)$ **fi.**

Note that the value of n is not needed after any inner call so it can be replaced by a global variable:

proc $\mathbf{A}_p(m, n) \;\equiv\; \mathbf{A}_1(m)$**.**
proc $\mathbf{A}_1(m) \;\equiv$
  if $m = 0$ **then** $r := n + 1$
  elsif $n = 0$ **then** $n := 1;\ \mathbf{A}_1(m - 1)$
    else $n := n - 1;\ \mathbf{A}_1(m);\ n := r;\ \mathbf{A}_1(m - 1)$ **fi.**

Within $\mathbf{A}_1$ we can use the same variable for n and r to get:

proc $\mathbf{A}_p(m, n) \;\equiv\; \mathbf{A}_1(m);\ r := n$**.**
proc $\mathbf{A}_1(m) \;\equiv$
  if $m = 0$ **then** $n := n + 1$
  elsif $n = 0$ **then** $n := 1;\ \mathbf{A}_1(m - 1)$
    else $n := n - 1;\ \mathbf{A}_1(m);\ \mathbf{A}_1(m - 1)$ **fi.**

The elimination of the statement $n := r$ between the two inner calls will be useful later.

The usual first step in recursion removal is to replace parameters by stacks. In this case it is not necessary since the initial value of the parameter m can be recovered after each recursive call and therefore can be preserved over the call of the procedure:

proc $\mathbf{A}_1(m) \;\equiv\; \mathbf{A}$**.**
$\mathbf{A} \;\equiv$ **if** $m = 0$ **then** $n := n + 1$
  elsif $n = 0$ **then** $n := 1;\ m := m - 1;\ \mathbf{A};\ m := m + 1$
    else $n := n - 1;\ \mathbf{A};\ m := m - 1;\ \mathbf{A};\ m := m + 1$ **fi.**

The inner procedure $\mathbf{A}$ preserves the value of m and sets n to $\mathbf{A}$(m, n). Noting that we want to increment m after the two inner calls at the ends of the second and third branches of the **if** , we transform $\mathbf{A}$ to a version which increments m rather than preserving it;

$\mathbf{A}_I \;\equiv$ **if** $m = 0$ **then** $n := n + 1;\ m := 1$
  elsif $n = 0$ **then** $n := 1;\ m := m - 1;\ \mathbf{A}_I$
    else $n := n - 1;\ \mathbf{A}_I;\ m := m - 1;\ \mathbf{A}_I$ **fi.**

Alternatively we may note that m is decremented *between* the two inner calls in the third line and so decide to alter $\mathbf{A}$ so that it decrements m:

$\mathbf{A}_D \;\equiv$ **if** $m = 0$ **then** $n := n + 1;\ m := -1$
  elsif $n = 0$ **then** $n := 1;\ m := m - 1;\ \mathbf{A}_D;\ m := m + 2$
    else $n := n - 1;\ \mathbf{A}_D;\ \mathbf{A}_D;\ m := m + 2$ **fi.**

We now discuss three general methods of recursion removal and apply them to the different recursive forms of Ackerman's function.

5 Method A: The "direct method"

The first method we shall discuss is the most often used method in which a protocol stack used to record the current state of the computation. This method is discussed in [3] where "actions" (parameterless procedure calls) are used as an intermediate step. The problem of recursion removal then reduces to the problem of removing non-terminal action calls. The first step is to translate $\mathbf{A}_1$ into an action system in which the non-terminal action calls are clearly displayed:

proc $\mathbf{A}_1(m) \;\equiv\; \mathbf{A};\ \mathbf{Z}$**.**
$\mathbf{A} \;\equiv$ **if** $m = 0$ **then** $n := n + 1;\ /\mathbf{A}$
  elsif $n = 0$ **then** $n := 1;\ \mathbf{B}$
    else $n := n - 1;\ \mathbf{A};\ \mathbf{B}$ **fi.**
$\mathbf{B} \;\equiv m := m - 1;\ \mathbf{A};$ **C** : .
  C : $\;\equiv m := m + 1;\ /\mathbf{A}$.
    $/\mathbf{A} \;\equiv$ **skip**.

Here $\mathbf{Z}$ is a statement which causes immediate termination of the outermost call of $\mathbf{A}_1$. In order to transform this to an iterative procedure we need to replace the sequence of operations (created by the calls of $\mathbf{A}$ in non-terminal positions) by an *explicit* sequence. We have therefore added the procedures $\mathbf{B}$, **C**: and $/\mathbf{A}$ to help with this. $/\mathbf{A}$ is called whenever a call of $\mathbf{A}$ terminates, it immediately terminates (since **skip** is a no-op) and control then passes to $\mathbf{Z}$, $\mathbf{B}$ or **C**: depending on whether the most recent activation of $\mathbf{A}$ to terminate was the outermost one, the first inner one or the second inner one respectively. We add a protocol stack on which this information will be recorded so that $/\mathbf{A}$ can read this information and call $\mathbf{Z}$, $\mathbf{B}$ or **C**: directly. This explicit call displaces the call which would occur if $/\mathbf{A}$ terminated. If the stack is empty then we call $\mathbf{Z}$, if the top element is 0 we call $\mathbf{B}$, and if it is 1 we call **C**: :

proc $\mathbf{A}_1(m) \ \equiv \ L := \langle\rangle;\ \mathbf{A}.$
$\mathbf{A} \ \equiv$ **if** $m = 0$ **then** $n := n + 1;\ /\mathbf{A}$
  elsif $n = 0$ **then** $n := 1;\ \mathbf{B}$
    else $n := n - 1;\ L \xleftarrow{\text{push}} 0;\ \mathbf{A}$ **fi.**
$\mathbf{B} \ \equiv m := m - 1;\ L \xleftarrow{\text{push}} 1;\ \mathbf{A}.$
  C : $\equiv m := m + 1;\ /\mathbf{A}.$
   $/\mathbf{A} \ \equiv$ **if** $L = \langle\rangle$ **then** $\mathbf{Z}$
     else $d \xleftarrow{\text{pop}} L$; **if** $d = 0$ **then** $\mathbf{B}$ **else C** : **fi fi.**

This transformation is proved for the general case in [19].

This results in a tail-recursive collection of procedures: we will now transform these into a single iterative procedure. Copy $\mathbf{B}$ into $\mathbf{A}$ and remove the recursion in $\mathbf{A}$:

$\mathbf{A} \ \equiv$ **do if** $m = 0$ **then** $n := n + 1$; **exit**
  elsif $n = 0$ **then** $n := 1;\ m := m - 1;\ L \xleftarrow{\text{push}} 1$
    else $n := n - 1;\ L \xleftarrow{\text{push}} 0$ **fi od**; $/\mathbf{A}.$

Copy **C**: and $\mathbf{B}$ into $/\mathbf{A}$ and then copy in $\mathbf{A}$ and remove the recursion:

$/\mathbf{A} \ \equiv$ **do if** $L = \langle\rangle$ **then exit fi**;
  $d \xleftarrow{\text{pop}} L$;
  if $d = 0$ **then** $m := m - 1;\ L \xleftarrow{\text{push}} 1$;
    do if $m = 0$ **then** $n := n + 1$; **exit**
     elsif $n = 0$ **then** $n := 1;\ m := m - 1;\ L \xleftarrow{\text{push}} 1$
       else $n := n - 1;\ L \xleftarrow{\text{push}} 0$ **fi od**
   else $m := m + 1$ **fi od**.

Take out the inner loop by double iteration:

$/\mathbf{A} \ \equiv$ **do do if** $L = \langle\rangle$ **then exit**(2) **fi**;
   $d \xleftarrow{\text{pop}} L$;
   if $d = 0$ **then** $m := m - 1;\ L \xleftarrow{\text{push}} 1$; **exit**
     else $m := m + 1$ **fi od**;
  do if $m = 0$ **then** $n := n + 1$; **exit**
   elsif $n = 0$ **then** $n := 1;\ m := m - 1;\ L \xleftarrow{\text{push}} 1$
     else $n := n - 1;\ L \xleftarrow{\text{push}} 0$ **fi od od**

Copy into $\mathbf{A}$ and apply proper inversion:

$\mathbf{A} \ \equiv$ **do do if** $m = 0$ **then** $n := n + 1$; **exit**
   elsif $n = 0$ **then** $n := 1;\ m := m - 1;\ L \xleftarrow{\text{push}} 1$
     else $n := n - 1;\ L \xleftarrow{\text{push}} 0$ **fi od**
  do if $L = \langle\rangle$ **then exit**(2) **fi**;
   $d \xleftarrow{\text{pop}} L$;
   if $d = 0$ **then** $m := m - 1;\ L \xleftarrow{\text{push}} 1$; **exit**

$\textbf{else } m := m + 1 \textbf{ fi od od}.$

Note that a stack of binary marks can be replaced by a single integer s, the digits in the binary representation of s represent the stack elements, with $s = 1$ representing the empty stack. Thus $d \stackrel{\text{pop}}{\longleftarrow} L$ is replaced by $\langle s, d\rangle := \langle s \div 2\rangle$ (which is the same as $d := (s \mod 2)$; $s := (s \div 2)$) and $L \stackrel{\text{push}}{\longleftarrow} t$ becomes $s := 2.s + t$. This will give us an iterative procedure for Ackerman's function which requires only a fixed amount of storage—which seems a remarkable achievement! However, we will show later that the maximum stack length is $A(m, n)$ and therefore the maximum size of the integer is $2^{A(m,n)}$ which will be very large even for small values of m.

5.1 Alternative Application of Method A

The direct method applied to $\mathbf{A}_D$ gives:

$$
\begin{array}{l}
\textbf{proc } \mathbf{A}_p(m, n) \ \equiv \ L := \langle\rangle;\ \mathbf{A}_D. \\
\mathbf{A}_D \ \equiv \textbf{if } m = 0 \textbf{ then } n := n + 1;\ m := -1;\ /\mathbf{A}_D \\
\qquad\quad \textbf{elsif } n = 0 \textbf{ then } n := 1;\ m := m - 1;\ L \stackrel{\text{push}}{\longleftarrow} 1;\ \mathbf{A}_D \\
\qquad\qquad\qquad \textbf{else } n := n - 1;\ L \stackrel{\text{push}}{\longleftarrow} 0;\ \mathbf{A}_D \textbf{ fi.} \\
/\mathbf{A}_D \ \equiv \textbf{if } L = \langle\rangle \textbf{ then } r := n;\ \mathbf{Z} \\
\qquad\qquad\qquad \textbf{else } \mathbf{d} \stackrel{\text{pop}}{\longleftarrow} L; \\
\qquad\qquad\qquad\quad \textbf{if } \mathbf{d} = 1 \textbf{ then } m := m + 1;\ /\mathbf{A}_D \\
\qquad\qquad\qquad\qquad\quad \textbf{else } L \stackrel{\text{push}}{\longleftarrow} 1;\ \mathbf{A}_D \textbf{ fi fi.}
\end{array}
$$

Replace the tail-recursions by loops (see [19] for the proofs of these transformations):

$$
\begin{array}{l}
\mathbf{A}_D \ \equiv \textbf{do while } m \neq 0 \textbf{ do} \\
\qquad\qquad \textbf{if } n = 0 \textbf{ then } n := 1;\ m := m - 1;\ L \stackrel{\text{push}}{\longleftarrow} 1 \\
\qquad\qquad\qquad \textbf{else } n := n - 1;\ L \stackrel{\text{push}}{\longleftarrow} 0 \textbf{ fi od}; \\
\qquad\quad n := n + 1;\ m := -1; \\
\qquad\quad \textbf{while } L \neq \langle\rangle \textbf{ do} \\
\qquad\qquad \mathbf{d} \stackrel{\text{push}}{\longleftarrow} L; \\
\qquad\qquad \textbf{if } \mathbf{d} = 1 \textbf{ then } m := m + 1 \\
\qquad\qquad\qquad \textbf{else } L \stackrel{\text{push}}{\longleftarrow} 1;\ \textbf{exit fi od od}.
\end{array}
$$

This is very similar to the direct method applied to $\mathbf{A}_I$.

6 Method B: The "postponed obligations" method

Here we record obligations to compute procedure calls or other statements on a stack together with the values they require. These obligations are "worked through" in turn until the stack is empty. The general case of a recursive procedure with two inner calls can be written in the form:

$$
\begin{array}{l}
\textbf{proc } F(x) \ \equiv \ \textbf{if B then } \mathbf{S}_1;\ F(g_1(x));\ \mathbf{S}_2;\ F(g_2(x));\ \mathbf{S}_3 \\
\qquad\qquad\qquad \textbf{else } \mathbf{S}_4 \textbf{ fi.}
\end{array}
$$

where $\mathbf{S}_1$, $\mathbf{S}_2$, $\mathbf{S}_3$, and $\mathbf{S}_4$ are statements which contain no calls to F.

If x is invariant over $\mathbf{S}_2$ this is equivalent to:

$$
\begin{array}{l}
\textbf{proc } F(x) \ \equiv \textbf{var } A := \langle 0, x\rangle; \\
\qquad\qquad \textbf{while } A \neq \langle\rangle \textbf{ do} \\
\qquad\qquad\quad \langle m, x\rangle \stackrel{\text{push}}{\longleftarrow} A; \\
\qquad\qquad\quad \textbf{if } m = 0 \to \textbf{if B then } \mathbf{S}_1;\ A \stackrel{\text{push}}{\longleftarrow} \langle 3, x\rangle;\ A \stackrel{\text{push}}{\longleftarrow} \langle 0, g_2(x)\rangle; \\
\qquad\qquad\qquad\qquad\qquad A \stackrel{\text{push}}{\longleftarrow} \langle 2, x\rangle;\ A \stackrel{\text{push}}{\longleftarrow} \langle 0, g_1(x)\rangle \\
\qquad\qquad\qquad\qquad \textbf{else } \mathbf{S}_4 \textbf{ fi} \\
\qquad\qquad\quad \square\ m = 2 \to \mathbf{S}_2 \\
\qquad\qquad\quad \square\ m = 3 \to \mathbf{S}_3 \textbf{ fi od.}
\end{array}
$$

where A and m are new variables local to the procedure. See [19] for the proof of this transformation.

Here the stack elements are pairs of the form $\langle d, v\rangle$ where d is 0 if a procedure call has been postponed, 2 if $\mathbf{S}_2$ has been postponed, and 3 if $\mathbf{S}_3$ has been postponed. v is the value which needs to be put in the variable x before the "obligation" (procedure call or statement execution) can be fulfilled.

Note that if, as in this case, the last mark pushed onto the stack on one line of the **if** is the same value as in the guard of that line, then we know that the next iteration will select that line so we can avoid the push and pop by adding an inner loop:

$$
\begin{array}{l}
\textsf{\textbf{proc}}\ F(x)\ \equiv \textsf{\textbf{var}}\ A := \langle 0, x\rangle :\\
\qquad\qquad \textsf{\textbf{while}}\ A \neq \langle\rangle\ \textsf{\textbf{do}}\\
\qquad\qquad\quad \langle m, x\rangle \xleftarrow{\text{pop}} A;\\
\qquad\qquad\quad \textsf{\textbf{if}}\ m = 0 \rightarrow \textsf{\textbf{while}}\ \mathbf{B}\ \textsf{\textbf{do}}\ \mathbf{S}_1;\ A \xleftarrow{\text{push}} \langle 3, x\rangle;\ A \xleftarrow{\text{push}} \langle 0, g_2(x)\rangle;\\
\qquad\qquad\qquad\qquad\qquad A \xleftarrow{\text{push}} \langle 2, x\rangle;\ x := g_1(x)\ \textsf{\textbf{od}};\ \mathbf{S}_4\\
\qquad\qquad\quad \square\ m = 2 \rightarrow \mathbf{S}_2\\
\qquad\qquad\quad \square\ m = 3 \rightarrow \mathbf{S}_3\ \textsf{\textbf{fi od.}}
\end{array}
$$

The transformation of a tail-recursive call into a **goto** statement (here replaced by a **while** loop) is discussed in [16].

In the case of Ackermann's function (version $\mathbf{A}_1$ above) we have $\mathbf{S}_2 = \mathbf{S}_3 =$ **skip** so we only need to postpone obligations to execute $\mathbf{A}$ and the mark can be dispensed with since all the marks have the value 0. (This was why we identified r and n above—otherwise $\mathbf{S}_2 = n := r$, which cannot be postponed so we would need the marks to indicate which statement had been postponed). We get:

$$
\begin{array}{l}
\textsf{\textbf{proc}}\ \mathbf{A}_1(m)\ \equiv\ L := \langle m\rangle;\ \mathbf{A}.\\
\mathbf{A}\ \equiv \textsf{\textbf{while}}\ L \neq \langle\rangle\ \textsf{\textbf{do}}\\
\qquad\quad m \xleftarrow{\text{pop}} L;\\
\qquad\quad \textsf{\textbf{if}}\ m = 0\ \textsf{\textbf{then}}\ n := n + 1\\
\qquad\quad \textsf{\textbf{elsif}}\ n = 0\ \textsf{\textbf{then}}\ n := 1;\ L \xleftarrow{\text{push}} (m-1)\\
\qquad\qquad\qquad\quad \textsf{\textbf{else}}\ n := n - 1;\ L \xleftarrow{\text{push}} (m-1);\ L \xleftarrow{\text{push}} m\ \textsf{\textbf{fi od.}}
\end{array}
$$

We will show later that the stack stores up to $A(m, n)$ values which range up to $A(m, n)$ in size, so if all stack positions are the same size this requires $A(m, n).\log_2 A(m, n)$ bits of store. The first method required only $A(m, n)$ bits for the stack.

To illustrate the operation of this program we will stack n as well as m at the end of each iteration of the loop, and immediately unstack it on the next iteration. Add another stack L':

$$
\begin{array}{l}
\textsf{\textbf{proc}}\ \mathbf{A}_1(m)\ \equiv\ L := \langle m\rangle;\ L' := \langle n\rangle;\ \mathbf{A}.\\
\mathbf{A}\ \equiv \textsf{\textbf{while}}\ L \neq \langle\rangle\ \textsf{\textbf{do}}\\
\qquad\quad m \xleftarrow{\text{pop}} L;\ n \xleftarrow{\text{pop}} L';\\
\qquad\quad \textsf{\textbf{if}}\ m = 0\ \textsf{\textbf{then}}\ n := n + 1;\ L' \xleftarrow{\text{push}} n\\
\qquad\quad \textsf{\textbf{elsif}}\ n = 0\ \textsf{\textbf{then}}\ n := 1;\ L \xleftarrow{\text{push}} (m-1);\ L' \xleftarrow{\text{push}} n\\
\qquad\qquad\qquad\quad \textsf{\textbf{else}}\ n := n - 1;\ L \xleftarrow{\text{push}} (m-1);\ L \xleftarrow{\text{push}} m;\ L' \xleftarrow{\text{push}} n\ \textsf{\textbf{fi od.}}
\end{array}
$$

Now combine L and L' into one stack L'' where $L'' = L +\!\!+\, L'$ and then replace L'' by L. We get:

$$
\begin{array}{l}
\textsf{\textbf{proc}}\ \mathbf{A}_1(m)\ \equiv\ L := \langle m, n\rangle;\ \mathbf{A}.\\
\mathbf{A}\ \equiv \textsf{\textbf{while}}\ L \neq \langle\rangle\ \textsf{\textbf{do}}\\
\qquad\quad \langle m, n\rangle \xleftarrow{\text{pop}} L;\\
\qquad\quad \textsf{\textbf{if}}\ m = 0\ \textsf{\textbf{then}}\ n := n + 1;\ L \xleftarrow{\text{push}} n\\
\qquad\quad \textsf{\textbf{elsif}}\ n = 0\ \textsf{\textbf{then}}\ n := 1;\ L := L +\!\!+ \langle m-1, n\rangle\\
\qquad\qquad\qquad\quad \textsf{\textbf{else}}\ n := n - 1;\ L := L +\!\!+ \langle m-1, m, n\rangle\ \textsf{\textbf{fi od.}}
\end{array}
$$

Push the statement $\langle m, n\rangle \xleftarrow{\text{pop}} L$ inside the **if** , $m = 0$ becomes $L[2] = 0$ etc. and we get:

proc $\mathbf{A}_1(m) \equiv$
  $L := \langle m, n\rangle$;
  while $\ell(L) \neq 1$ **do**
    if $L[2] = 0$ **then** $L := L[3\,.\,.] +\!\!+ \langle L[1] + 1\rangle$
    elsif $L[1] = 0$ **then** $L := L[3\,.\,.] +\!\!+ \langle L[2] - 1, 1\rangle$
      else $L := L[3\,.\,.] +\!\!+ \langle L[2] - 1, L[2], L[1] - 1\rangle$ **fi od**;
  $r := L[1]$.

It is easy to see intuitively that this terminates for every $n, m > 0$ and sets r to the value $A(m, n)$: the following invariant holds over the loop: If $L = \langle a_n, \dots, a_1\rangle$

$$A(m, n) = A(a_n, A(a_{n-1}, \dots, A(a_3, A(a_2, a_1)) \dots))$$

If we define the function Ack on sequences of integers as follows:

$$\begin{aligned} \mathsf{Ack}(L) &= A(a_n, A(a_{n-1}, \dots, A(a_3, A(a_2, a_1)) \dots)) \\ \mathsf{Ack}(\langle x\rangle) &= x \end{aligned}$$

then our invariant is $A(m, n) = \mathsf{Ack}(L)$. The body of the loop repeatedly applies the recursive definition of Ackerman's function to the rightmost two elements of the sequence L, and so preserves the invariant. On termination the length of L is one and so $\mathsf{Ack}(L) = L[1] = A(m, n)$, from the invariant, so $r = A(m, n)$. However, a direct proof of termination is not very easy, partly because of the extreme inefficiency of the procedure (which will be demonstrated later)—it is difficult to prove termination because the procedure very nearly doesn't terminate! We know that it *does* terminate because we derived it by transformation from a recursive function which we proved terminates. However it is instructive to try and devise a direct proof of termination (see below).

6.1 Alternative application of method B

We cannot apply the method of postponed obligations to $\mathbf{A}_I$, since the statement $m := m - 1$ between the inner calls alters the value of m which is used in the test. However the method *can* be applied to $\mathbf{A}_D$. Note that we do not have any parameters to record on the stack, but we do need to record whether $\mathbf{A}_D$ or the statement $m := m + 2$ has been postponed. Thus we need a binary stack. We get:

proc $\mathbf{A}_p(m, n) \equiv L := \langle 0\rangle;\ \mathbf{A}_D;\ r := n.$
$\mathbf{A}_D \equiv$ **while** $L \neq \langle\rangle$ **do**
    $d \xleftarrow{\text{pop}} L$;
    if $d = 1$ **then** $m := m + 2$
    elsif $m = 0$ **then** $n := n + 1;\ m := -1$
    elsif $n = 0$ **then** $n := 1;\ m := m - 1;\ L \xleftarrow{\text{push}} 1;\ L \xleftarrow{\text{push}} 0$
      else $n := n - 1;\ L \xleftarrow{\text{push}} 1;\ L \xleftarrow{\text{push}} 0;\ L \xleftarrow{\text{push}} 0$ **fi od.**

which by the transformation given above becomes:

proc $\mathbf{A}_p(m, n) \equiv L := \langle 0\rangle;\ \mathbf{A}_D;\ r := n.$
$\mathbf{A}_D \equiv$ **while** $L \neq \langle\rangle$ **do**
    $d \xleftarrow{\text{pop}} L$;
    if $d = 1$ **then** $m := m + 2$
      else while $m \neq 0$ **do**
        if $n = 0$ **then** $n := 1;\ m := m - 1;\ L \xleftarrow{\text{push}} 1$
          else $n := n - 1;\ L \xleftarrow{\text{push}} 1;\ L \xleftarrow{\text{push}} 0$ **fi od**;
      $n := n + 1;\ m := -1$ **fi od.**

7 Method C

The previous recursion removal methods are very general in their application. However, one problem with general methods is that using them it is not always possible to exploit regularities in the problem which can lead to more efficient algorithms. This is because a general method has to preserve the sequence of operations carried out by the recursive procedure. To get a more efficient version of the procedure we make use of the fact that in all the previous methods the function A is calculated many times over for many of the smaller pairs of values n and m. We can avoid this recalculation by maintaining a table of all the values calculated so far and then checking each pair of values to see if the result is in the table before attempting to calculate it. This method is discussed in [8], [7] and [1] which describes "Memorised" functions. (LISP functions which remember the results of all previous calls and can check if a new call is the same as a previous one: if it is then the function returns the stored value instead of re-calculating it). In our case $A(m, n) > 0$ for all m, n so we can initialise the table to zero and maintain the invariant: $T[m, n] > 0 \Rightarrow T[m, n] = A(m, n)$. Thus, before calculating a value we look in the table: if the value in the table is non-zero then we take the result from the table, if the table value is zero then we calculate the value and place the result in the table. We define a sub-procedure $\mathbf{A}_2(m, n)$ which when called will return with the value $T[m, n]$ correctly filled in. Thus if $\mathbf{A}_2(m, n)$ is called with $T[m, n]$ already filled in it has nothing to do.

proc $\mathbf{A}_p(m, n) \ \equiv\ T[*, *] := 0;\ \mathbf{A}_2(m, n);\ r := T[m, n]$**.**
proc $\mathbf{A}_2(m, n) \ \equiv$
  if $T[n, m] = 0$
    then if $m = 0$ **then** $T[m, n] := n + 1$
      elsif $n = 0$ **then** $\mathbf{A}_2(m - 1, 1);\ T[m, 0] := T[m - 1, 1]$
        else $\mathbf{A}_2(m, n - 1);\ \mathbf{A}_2(m - 1, T[m, n - 1]);$
          $T[m, n] := T[m - 1, T[m, n - 1]]$ **fi fi.**

Notice that to calculate $A(m, n)$ for $n > 0$ we need to calculate $A(m, n - 1)$, $A(m.n - 2)$, ..., $A(m, 0)$—thus the procedure will fill in the whole column from 0 to n (which has the value $A(m, n)$). We also need $A(m - 1, A(m, n - 1))$ so by a similar argument we see that the $m - 1$th column is filled in up to position $A(m, n - 1)$ which also has value $A(m, n)$. Continuing in this way we see that *all* the columns will be filled in until their final value is $A(m, n)$. A more direct way of doing this, which will also remove the recursion, is to start filling in column 1 one space at a time (using the fact that $A(1, n) = A(0, A(1, n - 1)) = A(1, n - 1) + 1$ and $A(1, 0) = A(0, 1) = 0 + 1 = 1$), after each step we fill in another space in each later column if this is possible: ie we keep the columns to the right of the first filled in as far as we are able. As soon as the nth position in the mth column is filled we know we have the required result.

If $\mathsf{pos}[i]$ is the last position filled for column i then we maintain:

$$2 \leqslant i \leqslant m \Rightarrow \big(\mathsf{pos}[i - 1] < T[i, \mathsf{pos}[i]] \ \vee\ \big(\mathsf{pos}[i] < 0\ \wedge\ \mathsf{pos}[i - 1] < 1\big)\big)$$

This can be done since whenever $\mathsf{pos}[i - 1] = T[i, \mathsf{pos}[i]]$ we know: $A(i, \mathsf{pos}[i] + 1) = A(i - 1, A(i, \mathsf{pos}[i])) = A(i - 1, T[i, \mathsf{pos}[i]]) = A(i - 1, \mathsf{pos}[i - 1]) = T[i - 1, \mathsf{pos}[i - 1]]$. so we can set $T[i, \mathsf{pos}[i] + 1] := T[i - 1, \mathsf{pos}[i - 1]]$ and $\mathsf{pos}[i] := \mathsf{pos}[i] + 1$ and maintain the invariant.

Also if we set $\mathsf{pos}[i - 1]$ to 1 then we know:

$$A(i, 0) = A(i - 1, 1) = A(i - 1, \mathsf{pos}[i - 1]) = T[i - 1, \mathsf{pos}[i - 1]].$$

so we can set $\mathsf{pos}[i] := 0$ and $T[i, \mathsf{pos}[i]] := T[i - 1, \mathsf{pos}[i - 1]]$.

This leads to the procedure:

proc $\mathbf{A}_p(m, n) \ \equiv$
  if $m = 0$ **then** $r := n + 1$
    else $T[*, *] := 0;\ \mathsf{pos}[*] := -1;\ T[0, 1] := 1;\ \mathsf{pos}[1] := 0;\ \mathbf{A}$ **fi.**

```
A ≡ while pos[m] ≠ n do
        pos[1] := pos[1] + 1; T[1, pos[1]] := T[1, pos[1] − 1] + 1;
        (fillinthenextpositionincolumn1);
        for i := 2 to m step 1 do
           (check if any of columns 2 to m can be extended);
           if pos[i − 1] = 1
             then pos[i] := 0; T[i, pos[i]] := T[i − 1, pos[i − 1]]
           elsif pos[i − 1] = T[i, pos[i]]
                then pos[i] := pos[i] + 1; T[i, pos[i]] := T[i − 1, pos[i − 1]] fi od od.
```

We can make the inner loop more efficient since once we have failed to extend a column (or if we extend a column to position 0 only) we will not be able to extend any of the columns to the right of it so we can terminate the inner loop.

Note also that all accesses to T are of the form $T[i, \mathsf{pos}[i]]$: only the final value in a column is accessed, earlier values being no longer needed. So we only need to store the final value in each column. We store these in an array $\mathsf{val}[*]$ where $\mathsf{val}[i] = T[i, \mathsf{pos}[i]]$ and then remove T from the program since it is never accessed. With these improvements our procedure becomes:

```
proc A_p(m, n) ≡
  if m = 0 then r := n + 1
        else val[*] := 0; pos[*] := −1; val[1] := 1; pos[1] := 0; A fi.
A ≡ while pos[m] ≠ n do
        pos[1] := pos[1] + 1; val[1] := val[1] + 1;
        i := 2;
        do if i > m then exit fi;
           if pos[i − 1] = 1
             then pos[i] := 0; val[i] := val[i − 1]; exit
           elsif pos[i − 1] = val[i]
                then pos[i] := pos[i] + 1; val[i] := val[i − 1]
                 else exit fi;
           i := i + 1; od od.
```

We now add another variable j which records how many columns have any values in them. This means that the initial assignments $\mathsf{val}[*] := 0$ and $\mathsf{pos}[*] := -1$ can be replaced by the simple assignment $j := 1$. We have the invariant:

$$1 \leqslant j \leqslant m \;\wedge\; \forall k.\, \big(1 \leqslant k \leqslant j \Rightarrow \mathsf{val}[k] = A(k, \mathsf{pos}[k])\big)$$

Note that within the inner loop we have:

$$\forall k.\, \big(2 \leqslant k \leqslant i \Rightarrow \mathsf{val}[k-1] = \mathsf{val}[k]\big)$$

so we can replace $\mathsf{val}[i] := \mathsf{val}[i-1]$ by $\mathsf{val}[i] := \mathsf{val}[1]$. (Representing $\mathsf{val}[1]$ by a scalar may make this assignment slightly more efficient). The result is:

```
proc A_p(m, n) ≡
  if m = 0 then r := n + 1
        else val[1] := 1; pos[1] := 0; j := 1; A fi.
A ≡ do if j = m then if pos[m] = n then exit fi fi;
      (to avoid accessing pos[m] until it has been assigned);
      pos[1] := pos[1] + 1; val[1] := val[1] + 1;
      i := 2;
      do if i > m then exit fi;
         if pos[i − 1] = 1
           then pos[i] := 0; j := i; val[i] := val[1]; exit
```

```
        elsif pos[i − 1] = val[i]
              then pos[i] := pos[i] + 1; val[i] := val[1]
               else exit fi;
        i := i + 1;  od od.
```

From this procedure we can prove directly (by induction on n) that:

$$\{m = 1\};\ \mathsf{val}[1] := 1;\ \mathsf{pos}[1] := 0;\ j := 1;\ \mathbf{A}$$
$$\approx\ \{m = 1\};\ j := 1;\ \mathsf{val}[1] := n + 2;\ \mathsf{pos}[1] := n$$

Taking out the $m = 1$ case therefore gives:

```
proc A(m, n) ≡
  if m = 0 then r := n + 1
  elsif m = 1 then r := n + 2
              else val[2] := 3; pos[2] := 0; j := 2; A fi.
A  ≡ do if j = m then if pos[m] = n then exit fi fi;
        pos[2] := pos[2] + 1; val[2] := val[2] + 2;
        i := 3;
        do if i > m then exit fi;
           if pos[i − 1] = 1
             then pos[i] := 0; j := i; val[i] := val[2]; exit
           elsif pos[i − 1] = val[i]
                 then pos[i] := pos[i] + 1; val[i] := val[2]
                  else exit fi;
           i := i + 1;  od od.
```

We can use *this* version to prove (again by induction on n) that:

$$\{m = 2\};\ \mathsf{val}[2] := 2;\ \mathsf{pos}[2] := 0;\ j := 2;\ \mathbf{A}$$
$$\approx\ \{m = 2\};\ j := 2;\ \mathsf{val}[2] := 2.n + 3;\ \mathsf{pos}[2] := n$$

Taking out the $m = 2$ case and $m = 3$ case in the same way we get:

```
proc A(m, n) ≡
  if m = 0 then r := n + 1
  elsif m = 1 then r := n + 2
  elsif m = 2 then r := 2.n + 3
  elsif m = 3 then r := 2^(n+3) − 3
              else val[4] := 13; pos[4] := 0; j := 4; A fi.
A  ≡ do if j = m then if pos[m] = n then exit fi fi;
        pos[4] := pos[4] + 1; val[4] := 2^(val[4]+3) − 3;
        i := 4;
        do if i > m then exit fi;
           if pos[i − 1] = 1
             then pos[i] := 0; j := i; val[i] := val[4]; exit
           elsif pos[i − 1] = val[i]
                 then pos[i] := pos[i] + 1; val[i] := val[4]
                  else exit fi;
           i := i + 1;  od od.
```

For $m > 4$ this version will calculate $A(m, n)$ in approximately $\mathsf{LOG}(A(m, n)) + 1$ steps where:

$$\mathsf{LOG}(x) = \text{the smallest } k \text{ st. } {}^{k}\log_2 x \leqslant 16$$
$$= \mu k.\big({}^{k}\log_2 x \leqslant 16\ \wedge\ \forall l > k.\,{}^{l}\log_2 x > 16\big)$$

that is: the number of applications of $\log_2$ to $A(m,n)$ required to bring it below 16. For example, this procedure will calculate $A(4,2) = 2^{65536} - 3$ in 3 steps (since $\log_2 2^{65536} = 65536$ and $\log_2 65536 = 16$; hence $\mathsf{LOG}(2^{65536} - 3) = 2$). Compare this with the approximately $2^{131073}/3$ steps required by method A. Similarly method E calculates $A(6,0) = {}^{65536}2 - 3$ in 65533 steps rather than about $2/3.({}^{65536}2)^2$ steps (see below for the justification of these figures). Even with modern high-speed computers this is a useful improvement!

8 Direct Proof of Termination

To derive a direct proof of termination we need to find a well-founded order relation on sequences such that L is decreased under this order by the execution of the body of the loop. However a simple lexical order on the elements of the sequence will not suffice.

If we ignore the (trivial) case $m = 0$ and $n = 0$ then it is easy to see that the invariant $\langle L[2], L[1]\rangle \neq \langle 0,0\rangle$ is maintained by the loop and since $A(0,n) = n+1$ and A is increasing in both arguments we have:

$$\mathsf{Ack}(L) \geqslant A(0, A(0, \dots, A(0,1)\dots)) = \ell(L) \quad (\ell(L) - 1 \text{ zeros})$$

This follows because if $L[1] = 0$ then we must have $L[2] > 0$ and so $A(L[2], L[1]) = A(L[2] - 1, 1) \geqslant A(0,1)$, all other cases follow from the monotonicity of A in both arguments.

So $\ell(L) \leqslant A(m_0, n_0)$ is invariant where m_0 and n_0 are the initial values of m and n.

Consider the final sequence of "pops" from the stack (ie the final sequence of iterations in which the first alternative is selected for execution each time), we claim that just before this sequence of pops the stack had the form $\langle 0, \dots, 0, 1\rangle$ with stack length A. It must be $\langle 0, \dots, 0, x\rangle$ with $x > 0$ since if say the $k+1$th element were non-zero, say y, then after k pops we would have $\langle 0, \dots, 0, y, x+k\rangle$ and the next iteration would be a "push". $x = 1$ since the only way to get a zero in $L[2]$ is by putting a 1 in $L[1]$ in the second alternative of the **if** .

$\mathsf{Ack}(L) = \ell(L)$ if L is of the form $\langle 0, \dots, 0, 1\rangle$ so the stack is A elements long at this point. It can never be more than A elements long because of the invariant above.

Our order relation on sequences will be a lexical order on the sequence of pairs:

$$\begin{aligned}\mathsf{Ord}(L) =_{\mathrm{DF}} \ & \langle\langle L(\ell(L)), \mathsf{Ack}(L \upharpoonright (\ell(L) - 1))\rangle, \\ & \langle L(\ell(L) - 1), \mathsf{Ack}(L \upharpoonright (\ell(L) - 2))\rangle, \\ & \langle L(\ell(L) - 2), \mathsf{Ack}(L \upharpoonright (\ell(L) - 3))\rangle, \\ & \dots\rangle\end{aligned}$$

So for example:

$$\begin{aligned}\mathsf{Ord}(\langle 3,4,5,6\rangle) = \ & \langle\langle 3, A(4, A(5,6))\rangle, \\ & \langle 4, A(5,6)\rangle, \\ & \langle 5,6\rangle, \\ & 0, 0, \dots\rangle\end{aligned}$$

where the sequence is filled out with zeros to have A elements and zero is considered less than any pair. Then our well-founded order on sequences is $L_1 \prec L_2$ iff $\mathsf{Ord}(L_1) < \mathsf{Ord}(L_2)$.

We now show that the execution of the loop body decreases L under this order: If $L = \langle x_k, \dots, x_1, 0, n\rangle$ where $k \geqslant 0$ then $L' = \langle x_k, \dots, x_1, n+1\rangle$ (the value of L after execution of the loop body) and $\ell(L') = \ell(L) - 1$. The first $k-1$ pairs of $\mathsf{Ord}(L)$ and $\mathsf{Ord}(L')$ are equal since

$$\mathsf{Ack}(\langle 0, n\rangle) = \mathsf{Ack}(\langle n+1\rangle)$$

The kth pairs are:

$$\begin{aligned}\mathsf{Ord}(L)[k] &= \langle L[3], \mathsf{Ack}(L \restriction 2)\rangle = \langle x_1, A(0,n)\rangle \\ &= \langle x_1, n+1\rangle = \langle L'[2], \mathsf{Ack}(L' \restriction 1)\rangle = \mathsf{Ord}(L)[k]\end{aligned}$$

The $k+1$th pairs are:

$$\begin{aligned}\mathsf{Ord}(L)[k+1] &= \langle L[2], \mathsf{Ack}(L \restriction 1)\rangle = \langle 0, n\rangle \\ &> 0 = \langle L'[1], \mathsf{Ack}(L' \restriction 0)\rangle = \mathsf{Ord}(L)[k+1]\end{aligned}$$

since $\mathsf{Ack}(L' \restriction 0)$ is undefined.

If $L = \langle x_k, \ldots, x_1, m, 0\rangle$ then $L' = \langle x_k, \ldots, x_1, m-1, 1\rangle$ and $\ell(L') = \ell(L)$ The first $k-1$ pairs of $\mathsf{Ord}(L)$ and $\mathsf{Ord}(L')$ are equal since

$$\mathsf{Ack}(\langle m, 0\rangle) = \mathsf{Ack}(\langle m-1, 1\rangle)$$

The kth pairs are:

$$\begin{aligned}\mathsf{Ord}(L)[k] &= \langle L[3], \mathsf{Ack}(L \restriction 2)\rangle = \langle x_1, A(m,0)\rangle \\ &= \langle x_1, A(m-1,1)\rangle = \langle L'[3], \mathsf{Ack}(L' \restriction 2)\rangle = \mathsf{Ord}(L)[k]\end{aligned}$$

The $k+1$th pairs are:

$$\begin{aligned}\mathsf{Ord}(L)[k+1] &= \langle L[2], \mathsf{Ack}(L \restriction 1)\rangle = \langle m, 0\rangle \\ &> \langle m-1, 1\rangle = \langle L'[2], \mathsf{Ack}(L' \restriction 1)\rangle = \mathsf{Ord}(L)[k+1]\end{aligned}$$

If $L = \langle x_k, \ldots, x_1, m, n\rangle$ then $L' = \langle x_k, \ldots, x_1, m-1, m, n-1\rangle$ and $\ell(L') = \ell(L) + 1$ The first $k-1$ pairs of $\mathsf{Ord}(L)$ and $\mathsf{Ord}(L')$ are equal since

$$\mathsf{Ack}(\langle m, n\rangle) = \mathsf{Ack}(\langle m-1, m, n-1\rangle).$$

The kth pairs are:

$$\begin{aligned}\mathsf{Ord}(L)[k] &= \langle L[3], \mathsf{Ack}(L \restriction 2)\rangle = \langle x_1, A(m,n)\rangle \\ &= \langle x_1, A(m-1, A(m, n-1))\rangle = \langle L'[4], \mathsf{Ack}(L' \restriction 3)\rangle = \mathsf{Ord}(L)[k]\end{aligned}$$

The $k+1$th pairs are:

$$\begin{aligned}\mathsf{Ord}(L)[k+1] &= \langle L[2], \mathsf{Ack}(L \restriction 1)\rangle = \langle m, n\rangle \\ &> \langle m-1, A(m, n-1)\rangle = \langle L'[3], \mathsf{Ack}(L' \restriction 2)\rangle = \mathsf{Ord}(L)[k+1]\end{aligned}$$

Incidentally, for the next pairs we have:

$$\begin{aligned}\mathsf{Ord}(L)[k+2] &= \langle L[1], \mathsf{Ack}(L \restriction 0)\rangle = 0 \\ &< \langle m, n-1\rangle = \langle L'[2], \mathsf{Ack}(L' \restriction 1)\rangle = \mathsf{Ord}(L)[k+2]\end{aligned}$$

but this doesn't matter because we are using a lexical order so the $k+1$th pairs take precedence.

Hence L is decreased and termination is proved.

If we also use the fact that $L(i) \leqslant A$ is an invariant, then the following integer function of L is decreased:

If $L = \langle a_n, \ldots, a_1\rangle$ then

$$\begin{aligned}t(L) = {} & A^{2A-4}a_n + A^{2A-5}\mathsf{Ack}(L \restriction (n-1)) + A^{2A-6}a_{n-1} + A^{2A-7}\mathsf{Ack}(L \restriction (n-2)) \\ & + \cdots + A^{2A-2n+2}a_3 + A^{2a-2n+1}A(a_2, a_1) + A^{2A-2n}a_1\end{aligned}$$

since $n \leqslant A$.

This also gives an upper bound for the number of iterations, namely $2A^{2A-3}$.

This small example shows one of the problems with the method of program development and verification using pre-and post-conditions as proposed in [14], [15] and [11]. The techniques can only demonstrate *partial* correctness (ie the program is correct provided it terminates). The proof of termination has to be carried out independently upon the final program, and this can require some ingenuity: as in the case above! A direct proof of termination (and of correctness) of the procedure arising from the first method (Method A) would appear to be much more difficult. Any contributions are welcome!

8.1 Determining the number of steps

To investigate the number of steps the iterative algorithm requires in more detail, we define $SA(m, n)$ to be the number of steps required to compute $A(m, n)$. This can be shown to be the same as the number of recursive calls the recursive procedure requires by inserting $c := c + 1$ at the beginning of each arm of the **if** . If $c = 0$ initially then the final value of c is the number of recursive calls. If we follow the statement through the transformations, we see that c is incremented once in the loop so the final value is the number of iterations. Hence $SA(m, n)$ is also the number of applications of the definition required to expand $A(m, n)$ into an integer.

Clearly $SA(0, n) = 1$, since the procedure terminates after a single iteration.

In computing $A(m, 0)$ with $m > 0$ the first iteration changes L from $\langle m, 0\rangle$ to $\langle m-1, 1\rangle$ which is then changed to $\langle A(m-1, 1)\rangle$in $SA(m-1, 1)$ steps ie

$$SA(m, 0) = SA(m-1, 1) + 1 \quad \text{for } m > 0$$

n	m: 0	1	2	3	4	5	6
0	1	2	3	5	13	65533	${}^{65536}2-3$
1	2	3	5	13	65533	${}^{65536}2-3$	
2	3	4	7	29	$2^{65536}-3$		
3	4	5	9	61	${}^{6}2-3$		
4	5	6	11	125	${}^{7}2-3$		
5	6	7	13	253	${}^{8}2-3$		
6	7	8	15	509	${}^{9}2-3$		
7	8	9	17	1021	${}^{10}2-3$		
8	9	10	19	2045	${}^{11}2-3$		
9	10	11	21	4093	${}^{12}2-3$		
10	11	12	23	8189	${}^{13}2-3$		
11	12	13	25	16381	${}^{15}2-3$		
12	13	14	27	32765	${}^{16}2-3$		
13	14	15	29	65533	${}^{17}2-3$		
$\vdots$							
n	$n+1$	$n+2$	$2n+3$	$2^{n+3}-3$	${}^{(n+3)}2-3$		

Figure 1: A Table of Values for $A(m, n)$

where ${}^{n}x = x^{x^{x^{\cdots x}}}$ (n times).

In computing $A(m, n)$ with $m > 0$, $n > 0$ the first iteration changes L from $\langle m, n\rangle$ to $\langle m-1, m, n-1\rangle$ which is then changed to $\langle m-1, A(m, n-1)\rangle$ in $SA(m, n-1)$ steps and thence to $\langle A(m-1, m, n-1)\rangle = \langle A(m, n)\rangle$ in $SA(m-1, A(m, n-1))$ steps so:

$$SA(m,n) = SA(m,n-1) + SA(m-1, A(m,n-1)) + 1 \quad \text{for } m > 0, n > 0$$

So we have the recursive definition:

$$\begin{aligned}
SA(0,n) &= 1 \\
SA(m,0) &= SA(m-1,1) + 1 \\
SA(m,n) &= SA(m,n-1) + SA(m-1, A(m,n-1)) + 1
\end{aligned}$$

From this definition we find that for $n > 0$:

$$\begin{aligned}
SA(1,n) &= SA(0,(n-1)+2) + SA(1,n-1) + 1 \\
&= SA(1,0) + 2n \\
&= 2(n+1)
\end{aligned}$$

$$\begin{aligned}
SA(2,n) &= SA(1, 2(n-1)+3) + SA(2,n-1) + 1 \\
&= SA(2,0) + \sum_{1\leqslant m\leqslant n} (4m+5) \\
&= 2n^2 + 7n + 5
\end{aligned}$$

$$\begin{aligned}
SA(3,n) &= SA(2, 2^{(n-1)+3} - 3) + SA(3,n-1) + 1 \\
&= \sum_{1\leqslant m\leqslant n} \left(2^{2m+5} - 5.2^{m+2} + 3\right) + SA(3,0) \\
&= 128/3.4^n - 40.2^n + 3n + 37/3
\end{aligned}$$

$$\begin{aligned}
SA(4,n) &= SA(3, A(4,n-1)) + SA(4,n-1) + 1 \\
&= SA(3, {}^{(n+2)}2 - 3) + 1 + SA(4,n-1) \\
&= 128/3.4^{{}^{(n+2)}2} - 40.2^{{}^{(n+2)}2} + 3({}^{(n+2)}2 - 3) + 37/3 + 1 + SA(4,n-1) \\
&= 2/3.({}^{(n+3)}2)^2 - 5.{}^{(n+3)}2 + 3.{}^{(n+2)}2 + 13/3 + SA(4,n-1) \\
&= 2/3.A(4,n)^2 - A(4,n) + 3.A(4,n-1) + 13/3 + SA(4,n-1)
\end{aligned}$$

For example:

$$\begin{aligned}
SA(4,1) &= SA(4,0) + SA(3, A(4,0)) + 1 \\
&= SA(3,1) + 1 + SA(3,13) + 1 \\
&= 108 + SA(3,13) = 108 + 128/3.4^{13} - 40.2^{13} + 3.13 + 37/3 = 2862984010 \\
SA(4,1) &= 2/3.A(4,1)^2 - A(4,1) + 3.A(4,0) + 13/3 + SA(4,0) \\
&= 2/3.65533^2 - 65533 + 3.13 + 13/3 + 107 = 2862984010
\end{aligned}$$

By induction on n:

$$\begin{aligned}
SA(4,n) &= \sum_{1\leqslant m\leqslant n} \left(2/3.A(4,m)^2 - A(4,m) + 3.A(4,m-1)\right) + 13/3.n + SA(4,0) \\
&= \sum_{0\leqslant k\leqslant n-1} \left(2/3.A(4,n-k)^2 + 2.A(4,n-k)\right) - 3.A(4,n) + 13/3.n + 146
\end{aligned}$$

$$A(4,n) = {}^{(n+3)}2 - 3 = 2^{{}^{(n+2)}2} - 3$$

so

$$\log_2(A(4,n)+3) = {}^{(n+2)}2 = A(4,n-1)+3$$

and

$$\log_2\log_2(A(4,n)+3) = {}^{(n+1)}2 = A(4,n-2)+3$$

if $n \geqslant 2$, and so on.

Define ${}^k\log_2 x \ =_{\rm DF}\ \log_2\log_2\ldots\log_2 x$ (k applications)

Then by induction on k we see that

$$\begin{aligned}
{}^k\log_2(A(4,n)+3) &= A(4,n-k)+3 \qquad \text{if } n \geqslant k \\
(A(4,n-k)+3)^2 &= A(4,n-k)^2 + 6.A(4,n-k)+9
\end{aligned}$$

so

$$A(4,n-k)^2 = ({}^k\log_2(A(4,n)+3))^2 - 6.{}^k\log_2(A(4,n)+3)+9$$

and

$$\begin{aligned}
SA(4,n) &= \sum_{0\leqslant k\leqslant n-1} \big(2/3.({}^k\log_2(A(4,n)+3))^2 - 4.{}^k\log_2(A(4,n)+3)+6 \\
&\qquad + 2.{}^k\log_2(A(4,n)+3)-6\big) - 3.A(4,n) + 13/3.n + 146 \\
&= \sum_{0\leqslant k\leqslant n-1} \big(2/3.{}^k\log_2(A(4,n)+3))^2 - 2.{}^k\log_2(A(4,n)+3)\big) \\
&\qquad - 3.A(4,n) + 13/3.n + 146
\end{aligned}$$

For $n \geqslant 2$ this means

$$\begin{aligned}
SA(4,n) &\approx 2/3.A(4,n)^2 - A(4,n) + 2/3.(\log_2(A(4,n)+3))^2 \\
&\qquad - 2.\log_2(A(4,n)+3) + 13/3.n + 146
\end{aligned}$$

to order $(\log_2\log_2 A(4,n))^2$ (this is exact for $n=2$)

Similarly:

$$\begin{aligned}
SA(5,n) &= 2/3. \sum_{0\leqslant k\leqslant A(5,n-1)-1} \big(({}^k\log_2(A(4,A(5,n-1))+3))^2 \\
&\qquad - 2.{}^k\log_2(A(4,A(5,n-1))+3)\big) \\
&\qquad - 3.A(4,A(5,n-1)) + 13/3.A(5,n-1) + 146 \\
&\qquad + SA(5,n-1)+1
\end{aligned}$$

Now $A(4,A(5,n-1)) = A(5,n)$ so this is

$$\begin{aligned}
&= 2/3. \sum_{0\leqslant k\leqslant A(5,n-1)-1} \big(({}^k\log_2(A(5,n)+3))^2 - 2.{}^k\log_2(A(5,n)+3)\big) \\
&\qquad - 3.A(5,n) + 5/3.A(5,n-1) + 146 \\
&\qquad + SA(5,n-1)+1
\end{aligned}$$

So by induction on n:

$$\begin{aligned}
SA(5,n) &= \sum_{1\leqslant m\leqslant n} \Big(\sum_{0\leqslant k\leqslant A(5,m-1)-1} \big(2/3.({}^k\log_2(A(5,m)+3))^2 - 2.{}^k\log_2(A(5,m)+3)\big) \\
&\qquad - 3.A(5,m) + 5/3.A(5,m-1)\Big) \\
&\qquad + 146n + 262984011
\end{aligned}$$

Finally by induction on m for $m \geqslant 4$ and $n \geqslant 1$:

$$SA(m,n) \approx 2/3.A(m,n)^2 - A(m,n) + 2/3.(\log_2(A(m,n)+3))^2 - 2.\log_2(A(m,n)+3)$$

to order $(\log_2\log_2 A(m,n))^2$

Proof: For $m \geqslant 4$, $n \geqslant 1$:

$$\begin{aligned}
SA(m+1,n) &= SA(m, A(m+1,n-1)) + SA(m+1,n-1)+1 \\
&\approx 2/3.A(m,A(m+1,n-1))^2 - A(m,A(m+1,n-1)) \\
&\quad + 2/3.(\log_2(A(m,A(m+1,n-1))+3))^2 \\
&\quad - 2.\log_2(A(m,A(m+1,n-1))+3) \\
&\quad + SA(m+1,n-1)+1
\end{aligned}$$

to order $(\log_2 \log_2 A(m, A(m+1,n-1)))^2$
(by induction hypothesis) $A(m+1,n) = A(m, A(m+1,n-1))$ so this is

$$\begin{aligned}
&\approx 2/3.A(m+1,n)^2 - A(m+1,n) \\
&\quad + 2/3.(\log_2(A(m+1,n)+3))^2 \\
&\quad - 2.\log_2(A(m+1,n)+3) \\
&\quad + SA(m+1,n-1)+1
\end{aligned}$$

to order $(\log_2 \log_2 A(m+1,n))^2$.

Now $SA(m+1,n-1)$ is of the order $A(m+1,n-1)^2$ which is negligible in comparison with $(\log_2 \log_2 A(m+1,n))^2$ (since $m+1>4$) so this term can be neglected. We get:

$$\begin{aligned}
SA(m+1,n) &\approx 2/3.A(m+1,n)^2 - A(m+1,n) + 2/3.(\log_2(A(m+1,n)+3))^2 \\
&\quad - 2.\log_2(A(m+1,n)+3)
\end{aligned}$$

to order $(\log_2 \log_2 A(m+1,n))^2$ as required.

n	m: 0	1	2	3	4	5
0	1	2	5	15	107	2862984011
1	1	4	14	106	2862984010	
2	1	6	27	541	$\sim 2^{131073}/3$	
3	1	8	44	2432		
4	1	10	65	10307		
5	1	12	90	42438		
6	1	14	119	172233		
7	1	16	152	693964		
8	1	18	189	2785999		
9	1	20	230	11164370		
10	1	22	275	44698325		
11	1	24	324	178875096		
12	1	26	377	715664091		
13	1	28	434	2862983902		
$\vdots$						
n	1	$2(n+1)$	$2n^2+7n+5$	$128/3.4^n - 40.2^n + 3n + 37/3$		

Figure 2: A Table of Values for $SA(m,n)$

As an example, to test the accuracy of our approximation for $SA(m,n)$, we calculate:

$$\begin{aligned}
A(4,1) &= 65533 \\
2/3.A(4,1)^2 &= 2863049392 + 2/3 \\
2/3.(\log_2(A(4,1)+3))^2 &= 2/3.(\log_2 65536)^2 = 2/3.16^2 = 512/3 = 170 + 2/3 \\
2\log_2(A(4,1)+3) &= 2.\log_2 65536 = 32
\end{aligned}$$

So

$$2/3.A(4,1)^2 - A(4,1) + 2/3.(\log_2(A(4,1)+3))^2 - 2.\log_2(A(4,1)+3) = 2862983998 + 1/3$$

So the error is $11 + 2/3$ and the answer is correct to 8 significant figures. Note that:

$$(\log_2 \log_2(A(4,1)+3))^2 = 16$$

so our estimate fits here as well.

This is because the extra terms (involving $\log_2$) which should not have been included with $m = 4$, $n = 1$ are nearly cancelled out by the terms $13/3.n + 146$ which were omitted.

$$\begin{aligned}
SA(4,2) &= SA(4,1) + SA(3, A(4,1)) + 1 \\
&= SA(4,1) + SA(3, 65533) + 1 \\
&= 2862984010 + 128/3.4^{65533} - 40.2^{65533} + 3.65533 + 37/3 + 1 \\
&= 128/3.4^{65533} - 40.2^{65533} + 2863180622 + 1/3
\end{aligned}$$

$$\begin{aligned}
A(4,2) &= 2^{65536} - 3 = 8.2^{65533} - 3 \\
2/3.A(4,2)^2 &= 2/3.(2^{65536} - 3)^2 \\
&= 128/3.4^{65533} - 32.2^{65533} + 6 \\
2/3.(\log_2(A(4,2)+3))^2 &= 2/3.(\log_2 2^{65536})^2 = 2/3.2^{32} \\
\log_2(A(4,2)+3) &= \log_2 2^{65536} = 65536
\end{aligned}$$

So

$$\begin{aligned}
&2/3.A(4,2)^2 - A(4,2) + 2/3.(\log_2(A(4,2)+3))^2 - 2.\log_2(A(4,2)+3) \\
&\quad = 128/3.4^{65533} - 40.2^{65533} + 2863180467 + 2/3
\end{aligned}$$

So the error is $154 + 2/3$ Note that $(\log_2 \log_2(A(4,2)+3))^2 = (\log_2 \log_2(2^{65536}))^2 = 256$ —so our estimate fits here as well.

For $m = 4$ with $n > 2$ and $m = 5$, with$n > 0$ and $m > 5$ for all n the term in $(\log_2 \log_2 A(m,n))^2$ is by far the largest neglected and has a coefficient less than one, so the error is smaller than this. As we have seen the error is also smaller than this for $m = 4$, $n = 1$, 2 and so for $m = 5$, $n = 0$, so to sum up:

The error is less than $(\log_2 \log_2 A(m,n))^2$ for $m = 4$, $n > 0$ and for $m > 4$ and all n.

For example $A(4,3) = {}^6 2 - 3$ so $SA(4,3)$ is roughly $2/3.({}^6 2)^2 = 2/3.(2^{2^{2^{2^{2^{2}}}}})^2 = 2/3.(2^{2^{65536}})^2 = 2/3.2^{2^{65537}}$ $\log_{10} SA(4,3) \;\approx\; \log_{10}(2/3) + 2^{65537} > 1 \cdot 2.10^{19728}$

Now $(\log_2 \log_2(A(4,3)+3))^2 = 65536^2 = 2^{32}$ (which has ten digits).

So our estimate will give $SA(4,3)$ correct to $1 \cdot 2.10^{19728} - 10$ significant digits.

To calculate the number of iterations around the inner loops for the first iterative procedure (Method A) we insert statements $c := c + 1$ in four places in the original procedure:

proc $\mathbf{A}_p(m,n) \;\equiv\; c := 0;\ \mathbf{A}_1(m);\ r := n.$
proc $\mathbf{A}_1(m) \;\equiv$
  if $m = 0$ **then** $c := c + 1;\ n := n + 1$
  elsif $n = 0$ **then** $c := c + 1;\ n := 1;\ \mathbf{A}_1(m - 1)$
    else $c := c + 1;\ n := n - 1;\ \mathbf{A}_1(m);\ \mathbf{A}_1(m - 1);\ c := c + 1;$ **fi.**

Following these through the transformations gives:

$$
\begin{array}{l}
\mathbf{A} \equiv \textbf{do if } m = 0 \textbf{ then } c := c+1;\ n := n+1;\ \textbf{exit} \\
\qquad \textbf{elsif } n = 0 \textbf{ then } c := c+1;\ n := 1;\ m := m-1;\ L \xleftarrow{\text{push}} 1 \\
\qquad\qquad \textbf{else } c := c+1;\ n := n-1;\ L \xleftarrow{\text{push}} 0 \textbf{ fi od} \\
\quad \textbf{do do if } L = \langle\rangle \textbf{ then exit}(2) \textbf{ fi}; \\
\qquad d \xleftarrow{\text{pop}} L; \\
\qquad \textbf{if } d = 0 \textbf{ then } m := m-1;\ L \xleftarrow{\text{push}} 1; \\
\qquad\qquad \textbf{do if } m = 0 \textbf{ then } c := c+1;\ n := n+1;\ \textbf{exit} \\
\qquad\qquad\quad \textbf{elsif } n = 0 \textbf{ then } c := c+1;\ n := 1;\ m := m-1;\ L \xleftarrow{\text{push}} 1 \\
\qquad\qquad\qquad \textbf{else } c := c+1;\ n := n-1;\ L \xleftarrow{\text{push}} 0 \textbf{ fi od} \\
\qquad\quad \textbf{else } c := c+1;\ m := m+1 \textbf{ fi od od.}
\end{array}
$$

where c is incremented for each innermost loop.

As for the other version we let $SB(m,n)$ be the number of steps (final value of c):

$$
\begin{aligned}
SB(0,n) &= 1 \\
SB(m,0) &= SB(m-1,1)+1 \\
SB(m,n) &= SB(m,n-1)+SA(m-1,A(m,n-1))+2
\end{aligned}
$$

The only difference is that the third equation has $+2$ instead of $+1$.

From this definition we get, for $n > 0$:

$$
\begin{aligned}
SB(1,n) &= 3n+2 \\
SB(2,n) &= 3n^2+10n+6 \\
SB(3,n) &= 3.2^{2n+4}-8.2^{n+2}+5+SB(3,n-1)
\end{aligned}
$$

So by induction on n:

$$
\begin{aligned}
SB(3,n) &= \sum_{1 \leqslant m \leqslant n} \left(3.2^{2m+4}-8.2^{m+2}+5\right)+SB(3,0) \\
&= 4^{n+3}-2^{n+6}+5n+20
\end{aligned}
$$

$$
\begin{aligned}
SB(4,n) &= SB(3,A(4,n-1))+SB(4,n-1)+2 \\
&= SB(3,{}^{(n+2)}2-3)+2+SB(4,n-1)
\end{aligned}
$$

By induction on n:

$$
\begin{aligned}
SB(4,n) &= \sum_{1 \leqslant m \leqslant n} \left(\left(A(4,m)+3\right)^2-8.\left(A(4,m)+3\right)+5.\left(A(4,m-1)+3\right)\right)+7.n+SB(4,0) \\
SB(4,n) &= \sum_{0 \leqslant k \leqslant n-1} \left(({}^k\log_2(A(4,n)+3))^2-3.{}^k\log_2(A(4,n)+3)\right) \\
&\quad -5.A(4,n)+7.n+219
\end{aligned}
$$

Finally by induction on m for $m \geqslant 4$ and $n \geqslant 1$ (as for SA):

$$
SB(m,n) \approx A(m,n)^2-2.A(m,n)+(\log_2(A(m,n)+3))^2-3.\log_2(A(m,n)+3)
$$

to within $(\log_2 \log_2 A(m,n))^2$.

The error in the estimate for $SB(4,1)$ is $(\log_2 65536)^2-3.\log_2(65536)-7-219=-18$ that is, the estimate is 18 less than it should be.

So even here it is no more than $(\log_2 \log_2(A(4,1)+3))^2+2$.

The estimate for $SB(4,1)$ is: $41+219=233$ more than it should be $(\log_2 \log_2(A(4,2)+3))^2 = 256$ so the estimate fits here also.

n	m: 0	1	2	3	4	5
0	1	2	6	20	154	4294443250
1	1	5	19	153	4294443249	
2	1	8	38	798	$\sim 2^{131073}/3$	
3	1	11	63	3619		
4	1	14	94	15400		
5	1	17	131	63533		
6	1	20	174	258098		
7	1	23	223	1040439		
8	1	26	278	4177980		
9	1	29	339	16744513		
10	1	32	406	67043398		
11	1	25	479	268304459		
12	1	28	558	1073479760		
13	1	41	643	4294443093		
$\vdots$						
n	1	$3.n+2$	$3n^2+10n+6$	$4^{n+3}-2^{n+6}+5n+20$		

Figure 3: A Table of Values for $SB(m, n)$

We can use this information to deduce how many times each alternative in the **if** statement of the second method is chosen, and hence how many times each elementary statement is executed for both methods. We add "counting statements" as follows:

proc $\mathbf{A}_p(m, n) \equiv$
  if $m = 0$ **then** $D := D + 1;\ r := n + 1$
  elsif $n = 0$ **then** $T := T + 1;\ \mathbf{A}_p(m-1, 1)$
    else $I := I + 1;\ J := J + 1;\ \mathbf{A}_p(m, n-1);\ \mathbf{A}_p(m-1, r)$ **fi.**

With these additions, the second method gives:

proc $\mathbf{A}_1(m) \ \equiv\ L := \langle m \rangle;\ \mathbf{A}.$
$\mathbf{A}$ $\equiv$ **while** $L \neq \langle\rangle$ **do**
  $m \xleftarrow{\text{pop}} L;$
  if $m = 0$ **then** $D := D + 1;\ n := n + 1$
  elsif $n = 0$ **then** $T := T + 1;\ n := 1;\ L \xleftarrow{\text{push}} (m-1)$
    else $I := I + 1;\ J := J + 1;\ n := n - 1;\ L \xleftarrow{\text{push}} (m-1);\ L \xleftarrow{\text{push}} m$ **fi od.**

Note that the stack length is decreased when the first alternative is chosen; it remains the same when the second is chosen and is increased when the third is chosen. Hence $\ell(L) = I - D$ is invariant over the loop. The loop terminates when $\ell(L) = 0$, so on termination we have $I = D$. The sum $D + T + I$ is increased exactly once on every loop, so on termination $D + T + I = SA(n, m)$. The sum $D + T + I + J$ is increased once when the first or second alternative is chosen and twice when the third is chosen, so $D + T + I + J = SB(n, m)$. I and J are increased together so $I = J$ is invariant. So if we let: $A = A(n, m) and B = \log_2 A(n, m)$ Then from above:

$$2.I + T = 2/3.A^2 - A + 2/3.B^2 - B \qquad (1)$$
$$3.I + T = A^2 - 2.A + B^2 - 3.B \qquad (2)$$

Hence:

$$I = 1/3.A^2 - A + 1/3.B^2 - B \qquad (2) - (1)$$
$$T = A + 3.B \qquad 3.(1) - 2.(2)$$

Inserting counting statements as follows:

proc $\mathbf{A}_p(m, n) \;\equiv$
  if $m = 0$ **then** $D := D + 1;\ r := n + 1$
  elsif $n = 0$ **then** $T := T + 1;\ \mathbf{A}_p(m - 1, 1)$
    else $I := I + 1;\ \mathbf{A}_p(m, n - 1);\ J := J + 1;\ \mathbf{A}_p(m - 1, r);\ K := K + 1$ **fi.**

(where $K = I$ throughout) and applying the first method gives:

$\mathbf{A} \equiv$ **do do if** $m = 0$ **then** $D := D + 1;\ n := n + 1;$ **exit**
  elsif $n = 0$ **then** $T := T + 1;\ n := 1;\ m := m - 1;\ L \xleftarrow{\text{push}} 1$
    else $I := I + 1;\ n := n - 1;\ L \xleftarrow{\text{push}} 0$ **fi od**
 do if $L = \langle\rangle$ **then exit**(2) **fi**;
  $d \xleftarrow{\text{pop}} L;$
  if $d = 0$ **then** $J := J + 1;\ m := m - 1;\ L \xleftarrow{\text{push}} 1;$ **exit**
   else $K := K + 1;\ m := m + 1$ **fi od od.**

So by using Kirchoff's law we can count how many times each statement is executed.

The totals for various kinds of statements are given in Figure 4

Statement	Method B	Method A	A − B
test variable	$3.I + 2.T$	$5.I + 2.T$	$2.I$
inc/dec variable	$3.I + T$	$4.I + T$	I
$n := 1$	T	T	0
test stack $= \langle\rangle$	$2.I + T$	$2.I$	$-T$
push	$2.I + T$	$2.I + T$	0
pop	$2.I + T$	$2.I + T$	0
exit	0	I	I

Figure 4: Execution Counts for Various Statement Types

9 Conclusion

In this paper we have illustrated several program transformations for recursion removal by applying them to the recursion inherent in Ackermann's function. Since many programs are most clearly specified using some form of recursion but most efficiently written using iterative constructs, it is very useful to have a set of proven transformations which will translate recursive specifications into efficient iterative algorithms. The transformation theory of [19] on which this paper is based has also proved valuable in the analysis of existing programs by transforming the program into a specification which is easier to understand and modify [18].

10 References

[1] Harold Abelson, Gerald Jay Sussman & Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[2] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen," *Math. Ann.* 99 (1928), 118–133.

[3] J. Arsac, "Syntactic Source to Source Program Transformations and Program Manipulation," *Comm. ACM* 22 (Jan., 1982), 43–54.

[4] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

[5] F. L. Bauer, *Programming as an Evolutionary Process*, Lect. Notes in Comp. Sci. #46, Springer-Verlag, New York–Heidelberg–Berlin, 1976.

[6] F. L. Bauer, "Program Development By Stepwise Transformations—the Project CIP," in *Program Construction*, G. Goos & H. Hartmanis, eds., Lect. Notes in Comp. Sci. #69, Springer-Verlag, New York–Heidelberg–Berlin, 1979, 237–266.

[7] F. L. Bauer & H. Wossner, *Algorithmic Language and Program Development*, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

[8] R. Bird, "Tabulation Techniques for Recursive Programs," *Comput. Surveys* 12 (1980), 403–417.

[9] G. V. Bochmann, "Multiple exits from a loop without the goto," *Comm. ACM* 16 (July, 1973), 443–444.

[10] P. A. Buhr, "A Case for Teaching Multi-exit Loops to Beginning Programmers," *SIGPLAN Notices* 20 (Nov., 1985), 14–22.

[11] D. Gries, *The Science of Programming*, Springer-Verlag, New York–Heidelberg–Berlin, 1981.

[12] M. Griffiths, *Program Production by Successive Transformation*, Lect. Notes in Comp. Sci. #46, Springer-Verlag, New York–Heidelberg–Berlin, 1976.

[13] M. Griffiths, *Development of the Schorr-Waite Algorithm*, Lect. Notes in Comp. Sci. #69, Springer-Verlag, New York–Heidelberg–Berlin, 1979.

[14] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM* (1969).

[15] C. A. R. Hoare, "Procedures and parameters: An axiomatic approach," in *Symposium on Semantics of Algorithmic Languages*, E. Engeler, ed., Lect. Notes in Math. #188, Springer-Verlag, New York–Heidelberg–Berlin, 1971, 102–116.

[16] D. E. Knuth, "Structured Programming with the GOTO Statement," *Comput. Surveys* 6 (1974), 261–301.

[17] D. Taylor, "An Alternative to Current Looping Syntax," *SIGPLAN Notices* 19 (Dec., 1984), 48–53.

[18] M. Ward, "Transforming a Program into a Specification," Durham University, Technical Report 88/1, 1988, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/TR-88-1.ps.gz`⟩.

[19] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.