Generalizing Cook's Transformation to Imperative Stack Programs

Nils Andersen and Neil D. Jones

Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark

Abstract. Cook's construction from 1971 [4] shows that any two-way deterministic pushdown automaton (2DPDA) can be simulated in time $\mathcal{O}(n)$, where n is the length of its input string, and the more general [5] describes analogous results for other abstract machines. The goal of this paper is to make Cook's result usable for a broader spectrum of practical problems.

We introduce a family of one-stack programs that includes 2DPDAs, and present a uniform way to *compile* any imperative stack program into a new and often faster version by using memoization. The method only explores the computational configurations that are *reachable on the current input*, in contrast to Cook's method, and builds programs that run in linear time if the original was a 2DPDA in program form. The transformation applies to algorithms not previously dealt with by Cook's method, *e.g.* fast computation of functions such as Fibonacci and $\binom{m}{n}$.

1 Introduction

Stephen A. Cook described a transformation in 1971 that can, for instance, improve program running times from exponential to linear (as functions of their input size). This is interesting in that it delineates a class of programs can be simulated *faster than they run*, by using a richer storage structure for memoization.

Cook's result inspired the now widely used Knuth-Morris-Pratt string matching algorithm; an example where a theoretical insight led to practically useful techniques.

The original formulations [4,5][1, Section 9.4] were in terms of a class of automata and dealt with deciding membership in formal languages, *i.e.* sets of strings, but work equally well on programs.

The method involves application of offline memoization, building a table to save the results of intermediate computational configurations, and so to avoid recomputation. The memoization table size is input-dependent, sometimes large but always linear in the number of the program's "surface configurations".

A surface configuration consists of a program point, the contents of the top of the stack, and the current values of all program variables (except the stack); the method is therefore particularly interesting for algorithms with a small number and range of program variables. Further, it is not necessary to count as part of the surface configuration variables of two kinds: those that do not vary after program

initialization (*e.g.* regarding the input); and those that do not influence the control flow of the algorithm. We treat the former kind of variables as constants, and to avoid variables of the latter kind this paper uses a trick: instead of operations that only push or pop one item to or from the stack we allow a very general kind of stack operation. An operation of *arity* (a, b) will remove $a + 1$ elements from the stack and use them to determine $b + 1$ elements to be pushed.

Cook's table construction in effect *interprets* the program in an indirect way, and is general enough to handle deterministic and nondeterministic programs. This involves a nontrivial "bookkeeping" overhead. From an application viewpoint, another disadvantage is that table entries are made in a "speculative" or offline way, even if the program is deterministic — so that in practice most table entries turn out to be for computational configurations the program can never enter. Jones [7] (assuming the program to be deterministic) modifies Cook's approach to build the table online during simulation so that only the table entries actually needed are constructed, but is still in essence interpretive.

Bird [3] extends the transformation to computations with general values as input and output, as we also will do. However, Bird only treats programs of a very special form (built around a loop with a single pop instruction), and the memoizing transformation has to be done by hand.

Amtoft *et al.* [2] implement the method of [7], and use partial evaluation [8] to reduce the earlier methods' interpretive overhead [4,1,7]. During the development of this paper it was found that many of the proofs have strong similarities to ideas in [2]. Differences: our methods produce yet more efficient programs, and handle a significantly larger program class.

The present paper demonstrates a general method to *compile* a stack program directly into an equivalent online memoizing program that runs in linear time if the original was a 2DPDA in program form. It has no interpretive overhead at all, and significantly less other bookkeeping than any of the methods just mentioned, thus bringing Cook's general result closer to practical usability. As a by-product, the method also yields a proof of Cook's construction that is more perspicuous than the original (in [4,1,7]).

In Section 2 program notation and semantics is introduced, and Section 3 describes the transformation. In Section 4 the transformation is proven to be faithful, and the output program's running time is analyzed and proven to be proportional to a certain value computable from the program text. If the program uses only ordinary push and pop instructions, this value is at most the number of "surface configurations". If more complex stack operations are employed, the values pushed onto the stack and the amount by which the stack may increase also enter into the value.

A program to find the longest palindromic prefix of a string is used to illustrate the transformation, and in Sections 5 and 6 two further examples, the *subsequence problem* and computation of *binomial coefficients* are presented. In the latter cases our method improves two exponential methods to ones whose execution times are bounded by products.

We conclude with some considerations of the applicability of the method.

2 Stack programs

The following is formulated in a way closer to daily programming practice than Cook's 2DPDA or auxiliary pushdown automata [4,5]. One difference is the use of an imperative programming style rather than transitions by sets of tuples as traditional in automata theory. Another difference is that the stack alphabet T is not necessarily finite, *e.g.* one may store integers there for future retrieval.

Notation: [] denotes the empty stack, $A : S$ is the result of pushing A onto the stack S, and $\mathrm{top}(S)$ is the topmost element of a non-empty stack S. List notation $[A_n, \ldots, A_2, A_1]$ denotes $A_n : \ldots : A_2 : A_1 : []$, stack $S{+\!\!+}S'$ is stack S on top of (appended to) stack S', and in such a case S' is said to be a *bottom* of $S{+\!\!+}S'$. For a natural number i, $i{\downarrow}S$ selects the i topmost elements of S, so $i{\downarrow}(A_n : \ldots : A_2 : A_1 : []) = A_n : \ldots : A_{n-i+1} : []$. The depth of S is written $|S|$, so $|[A_n, \ldots, A_1]| = n$.

For technical convenience we will generally assume that the "auxiliary store" or input tape scanning position is contained in the topmost stack element, but will on occasion break this convention by using an explicit *memory*.

2.1 Syntax

A *stack program* is a flow chart built with a collection of program statements labeled by a set L of *labels* $j, k, j', \ldots$ with a designated *initial label* j_0. Program statements refer to a *stack* S whose items are drawn from the *stack alphabet* T, which also contains the *bottom marker* Z.

A *stack function of arity* (a, b) is a function from T^{a+1} to T^{b+1}, for natural numbers a and b. Its use is to pop $a+1$ stack items and then to push $b+1$ values in their place. Thus a stack function of arity $(0, 0)$ updates the stack top only, by convention containing the memory and/or input tape scanning position.

Programs have no read statements, so input must be coded into the memory, to be manipulated by predicates and stack functions during execution. The dependency on input does not appear explicitly in our program notation.

A stack program consists of an initializing program statement (init) and a mapping, associating with each program label j a unique statement which is either a test (test), a stack operation of arity (a, b) ($\mathsf{stackop}_{a,b}$), or a terminal statement (term):

(init)	`begin` S `:=` $[Z]$`; go_to` j_0
(test)	j`: if` $p(\mathrm{top}(S))$ `then go_to` k^+ `else go_to` k^-
($\mathsf{stackop}_{a,b}$)	j`:` S `:=`f`=` S`; go_to` k
(term)	j`: end`

where j, k, k^+ and k^- are labels in L, p is a predicate over T, a and b are positive integers, and f is a stack function of arity (a, b).

A $\mathsf{stackop}_{0,0}$ statement amounts to an ordinary assignment, a $\mathsf{stackop}_{0,b}$ statement where $b > 0$ is a *push* statement of arity b, and a $\mathsf{stackop}_{a,0}$ statement with

$a > 0$ is a *pop* statement of arity a. Many programs will only use $\mathsf{stackop}_{a,b}$ statements where a or b (or both) are 0; but the use of other combinations can yield more efficient transformed programs, as will be seen in the Fibonacci and $\binom{m}{n}$ examples.

2.2 Semantics

A *program configuration* is its "total state", a pair (j, S) where j is the control point, and S is the current stack contents. A *surface configuration* is a pair (j, A_1) where $(j, [A_1, \ldots, A_n])$ is a program configuration.

Each stack program gives rise to a *next state relation* " " on its set of program configurations. For technical reasons the relation is defined as ternary, also taking a stack argument. The significance of $(j, S) \quad_{S'} (j_1, S_1)$ is going to be "execution of the statement at label j with stack S leads in one step to label j_1 and new stack S_1 without using S' (which is a bottom part of both S and S_1)".

For a test statement

$$j\texttt{: if } p(\mathrm{top}(S))\texttt{ then go\_to } k^{+}\texttt{ else go\_to } k^{-}$$

define (for every bottom S' of S)

$$(j, A : S) \quad_{S'} (k^{+}, A : S)$$

if $p(A)$ holds, else

$$(j, A : S) \quad_{S'} (k^{-}, A : S)$$

In the case of a $\mathsf{stackop}_{a,b}$ statement

$$j\texttt{: } S\texttt{ :=}f\texttt{= } S\texttt{; go\_to } k$$

define

$$(j, A_a : \ldots : A_1 : A_0 : S) \quad_{S'} (k, B_b : \ldots : B_1 : B_0 : S)$$

where S' is any bottom of S, and $f(A_0, A_1, \ldots, A_a) = (B_0, B_1, \ldots, B_b)$. A small but necessary point: execution is assumed to terminate abnormally if a $\mathsf{stackop}_{a,b}$ statement is reached with fewer than $a + 1$ elements on the stack or if a test statement is reached with an empty stack.

Notation "$\texttt{push } A \texttt{ onto } S$" stands for "$S\texttt{ :=}f\texttt{= } S$" where f is of arity $(0, 1)$, and $f(A_0) = (A_0, A)$. (Remark: A may be a function of A_0.) Similarly "$\texttt{pop } S$" stands for "$S\texttt{ :=}f\texttt{= } S$" with f of arity $(1, 0)$, and $f(A_0, A_1) = (A_0)$[1].

The *multiple step state transition* relation is the reflexive transitive closure $\overset{*}{}_{S}$ of $\quad_{S}$, and $\overset{+}{}_{S}$ is its transitive closure. Symbols $\quad$, $\overset{+}{}$ and $\overset{*}{}$ denote $\overset{}{}_{[]}$, $\overset{+}{}_{[]}$ and $\overset{*}{}_{[]}$. For a given input, the *computation* with the stack program is the sequence

$$(j_0, [Z]) = (j_0, S_0) \quad (j_1, S_1) \quad \ldots \quad (j_n, S_n) \quad \ldots$$

[1] Note that these do not change A_0, in keeping with the convention that A_0 is the auxiliary memory, containing variables such as the input scanning position.

A computation may be infinite, or it may end if a term statement is met. In the latter case it is called a *terminating computation*, and the *length* of such a computation is the number of program configurations that it contains. The pairs (j, S_j) are called *reachable configurations*.

A stack program obviously has a unique computation, *i.e.* it is *deterministic* (the next state relation is a partial function).

Our main result is that any terminating stack program may be compiled into another *whose run time is linear in the number of reachable surface configurations*. This number may be much less than the number of *all* computational configurations (involving the stack) entered by the program in its computation, and it is in no case larger than $\#L \cdot \#T$ (the bound obtained by Cook).

2.3 Notational extensions

It is often convenient to work with ordinary program variables in addition to the stack, so a more detailed total configuration could be a triple (j, mem, S) with a state of the *memory* in addition to the previous components. The way our machinery deals with such an extension is to consider the memory to be part of the top of the stack. Changing the values of such ordinary program variables is done by an operation with a stack function of arity $(0, 0)$. Formally, therefore, the described situation is covered by a suitable extension of the stack alphabet T (although the extended alphabet is only used in the top of the stack and not relevant for items buried deeper in the stack).

In practical examples, we shall freely use ordinary variables, with an understanding of the underlying formal model as described. It will later become necessary to consider other variants, containing extra stacks, tables, *etc.* These can be fit into the framework just given by further extensions of the memory state set.

An example. Let us, as an example, consider the problem of finding the length of the longest palindromic prefix of a given string $t_1 t_2 \ldots t_n$, where each $t_i \in T_0 \backslash \{Z\}$ and T_0 is a fixed finite alphabet. (Since the result is a number, this problem is more general than a decision problem.) The problem may be solved by a stack algorithm in the following way: for decreasing values of i, $i = n, n-1, n-2, \ldots$, reversed prefixes $t_i \ldots t_2 t_1$, kept on the stack, are compared to the given string. During a comparison matching symbols are popped, but after a mismatch the stack can be restored by means of the given string, and the next shorter prefix can be tried.

This naïve approach has been programmed in Figure 1 where a PASCAL-like notation is used, rather than the strict formalism with labeled statements and explicit jumps. An ordinary variable i as explained above is also used.

The input consists of n and the current string t and has been coded into the program. The program may run in time $\mathcal{O}(n^2)$ due to the backing up needed for unsuccessful partial matches, exemplified by strings of the form $A^p B A^{3p}$.

```
S := [Z];
i := 0;
while i < n do begin
      i := i + 1;
      push t_i onto S
end;
{S = [t_n, ..., t_1, Z]}
i := 0;
while top(S) ≠ Z do begin {invariant: S = [t_{h-i}, ..., t_1, Z], t_h ... t_{h-i+1} = t_1 ... t_i}
      if top(S) = t_{i+1} then i := i + 1
      else {restore} while i > 0 do begin
            push t_i onto S;
            i := i - 1
      end;
      pop S
end;
write("length of longest palindromic prefix is ",i)
```

Fig. 1. Naïve stack program to find for a given string $t_1 t_2 \dots t_n$ the largest i such that $t_1 t_2 \dots t_i$ is equal to its own reversal $t_i \dots t_2 t_1$.

The variable i assumes values between 0 and n, so the size of the total alphabet T is $(\#T_0 + 1) \cdot (n + 1)$, and we shall see that the program can be simulated in a number of steps proportional to this value.

3 The improvement

The crucial observation (of both Cook's work and our own) is that in a computation, the entire series of configurations following any total configuration $(j, A : S)$ is determined by *the surface configuration* (j, A) *alone*, until (if ever) some symbol deeper in the stack than A is used or popped. Stated more formally:

For all stacks S the following equivalence holds:

$$(j, [A]) \quad ^* (j', [A']) \text{ if and only if } (j, A : S) \quad ^*_S (j', A' : S)$$

Thus any two steps that lead to the same configuration $(j, A : S)$ will repeat the same subcomputation, until (if ever) some symbol from S is used or popped, *i.e.* the subcomputation is functionally determined by surface configuration (j, A).

This may be exploited to optimize the program. Suppose $(j, A : S) \quad ^*_S (j', S' +\!\!+ S)$ where j' is a $\mathsf{stackop}_{a,b}$ statement with $a \geq |S'|$. The first time $(j, A : S)$ is encountered, the program is run until $(j', S' +\!\!+ S)$ is entered. The surface part of this configuration is called the "terminator" in [1].

The pair (j', S') can then be stored for future reference, and if ever a configuration $(j, A : S_1)$ is entered again (for any S_1 at all), an immediate "short cut" can be taken to $(j', S' +\!\!+ S_1)$.

To do this we will add to the program a partial mapping from surface configurations, a table $dest : L \times T \to L \times T^*$, to remember the terminators (j', S'). Implementation note: *dest* could be implemented by a hash table, so that the memory required need only be of the order of the number of surface configurations actually entered.

The following section works out the details and shows the new "compilation" technique.

3.1 Transformation

To keep track of surface configurations that have been met, but whose terminators have not yet been found, an auxiliary stack *dump* is introduced; it will be driven in lockstep with S. Each entry in *dump* is a list of surface configurations, so *dump* is a list of lists.

We now modify `pgm` to give program `pgm′` that will, whenever a $\mathsf{stackop}_{a,b}$ instruction with $a > 0$ is encountered, store the terminators of those surface configurations whose subcomputations have been completed. Further, `pgm′` may consult the *dest* table to take a "shortcut" when a surface configuration is encountered whose terminator has already been computed. Let $dest_0$ denote the totally undefined mapping.

As a result, computations by `pgm′` are not in a one-to-one correspondence with those of `pgm`, but will avoid sometimes quite long recomputations.

At this stage only potential locations of the shortcuts are indicated, postponing the decisions as to which ones actually must be present.

The individual program statements of `pgm` are transformed as shown below, using the same labels and command forms in `pgm′` as in `pgm`.

(init) `begin` S `:=` $[Z]$`;` *dump* `:=` $[[]]$`;` *dest* `:=` $dest_0$`;` `go_to` j_0

(test) j: potential shortcut`;`
`if` $p(\mathrm{top}(S))$ `then go_to` k^+ `else go_to` k^-

($\mathsf{stackop}_{0,b}$) j: potential shortcut`;` S `:=`f`=` S`;`
`for` i`:=1 to` b `do push` [] `onto` *dump*`;` `go_to` k

($\mathsf{stackop}_{a,b}$) where $a > 0$
j: `for` i`:=1 to` a `do begin`
`for` each (j', A') on the list top(*dump*) `do`
dest(j', A') `:=` $(j, i{\downarrow}S)$`;`
`pop` this list from *dump*
`end;`
S `:=`f`=` S`;`
`for` i`:=1 to` b `do push` [] `onto` *dump*`;` `go_to` k

(term) j: `end`

At the positions "potential shortcut;" in the table above one may or may not insert a call "*shortcut*$(j, \mathrm{top}(S))$;" activating the following program segment

shortcut$(j, A) \equiv$
    `if` *dest*(j, A) `=` (j', S') `then begin`
        `pop` S`; push` the symbols of S' `onto` S`;`
        `push` $\#S' - 1$ empty lists `onto` *dump*`; go_to` j'
    `end else` {*dest*(j, A) is undefined}
        top(*dump*) `:=` (j, A) : top(*dump*)

Strictly speaking, the construction "`go_to` j" where j is found by computation (as used in the program segment above) extends our program notation. It amounts to a Fortran "computed goto", or can be realized by insertion of a series of tests and jumps to statically known program points. (Note that this transformation is completely determined by the program text and thus only influences running time by a constant factor.)

The proof that the transformed program is as desired has two sides: it must be proven that it is a faithful simulation of the original one, and that it executes in linear time (in a sense later made precise).

4 Proofs

4.1 Faithfulness

Total configurations of `pgm′` are quadruples of the form $(j, S, dump, dest)$. Symbols $\;$ and * are used for the next state relation and the multiple step state transition, respectively, between these new total states. Recall that $dest_0$ is the totally undefined mapping.

Lemma 1 `pgm′` only simulates `pgm` actions. *Assume that, in* `pgm′`*:*

$$(j_0, [Z], [[]], dest_0) \;^* (j, S, dump, dest)$$

Then

(a) `pgm` would do the same:
$$(j_0, [Z]) \;^* (j, S)$$

(b) entries in *dest* reflect subcomputations:
if $dest(j', A') = (j'', S'')$*, then* $(j', [A']) \;^+ (j'', S'')$*, and* j'' *labels a* $\mathsf{stackop}_{a,b}$ *statement with* $a > 0$

(c) information in *dump* is as intended:
if some pair (j', A') *is present in one of the lists on dump, say* $dump = dump'' +\!\!+ [[\ldots, (j', A'), \ldots]] +\!\!+ dump'$*, and* S' *is the bottom of the stack corresponding to* $dump'$*,* $S = S'' +\!\!+ S'$ *where* $|S'| = \left|dump'\right|$*, then*

$$(j_0, [Z]) \;^* (j', A' : S') \;^+_{S'} (j, S)$$

Proof. The three claims are proved simultaneously, by induction on the length of the computation in $\texttt{pgm}'$. First, all hold trivially for 0-step computations. Now consider

$$(j_0, [Z], [[]], \mathit{dest}_0) \ ^* \ (j_1, S_1, \mathit{dump}_1, \mathit{dest}_1) \quad (j, S, \mathit{dump}, \mathit{dest})$$

and assume (a), (b) and (c) of the computation leading to $(j_1, S_1, \mathit{dump}_1, \mathit{dest}_1)$.

If the transition in $\texttt{pgm}'$ from j_1 to j does not take a shortcut, (a) follows immediately; if a shortcut is taken, (a) follows inductively from (b).

Assume $\mathit{dest}(j', A') = (j'', S'')$. If $\mathit{dest}_1(j', A')$ was already defined, (b) holds. If not, $\mathit{dest}(j', A')$ must be defined by the statement at j_1. In that case $j_1 = j''$, j_1 must label a $\mathsf{stackop}_{a,b}$ statement with $a > 0$, and (b) follows inductively from (c).

Finally, assume the premises of (c). If (j', A') was already present in dump_1, the conclusion follows as in the proof of (a) above. If (j', A') is added by the statement at j_1 we must have $j_1 = j'$, and the statement at j_1 must contain the call "$\mathit{shortcut}(j_1, \mathrm{top}(S_1))$". In that case the conclusion of (c) follows inductively from (a). □

Lemma 2. *If a computation with* $\texttt{pgm}'$ *inserts some pair* (j, A) *more than once into dump, then it does not terminate.*

Proof. Once a pair (j, A) is removed from *dump*, $\mathit{dest}(j, A)$ becomes defined, preventing (j, A) from ever being entered into *dump* again. If, on the other hand, (j, A) is inserted into a *dump* that contains this pair already, Lemma 1(c) implies that the computation will continue forever. □

The converse result is valid in the following form:

Lemma 3 $\texttt{pgm}'$ **simulates all** $\texttt{pgm}$ **actions.** *If*

$$(j_0, [Z]) \ ^* \ (j, S)$$

then there will exist $S', \mathit{dump}', \mathit{dest}'$ *such that*

$$(j, S) \ \stackrel{*}{_{S'}} \ (j', S') \text{ and}$$
$$(j_0, [Z], [[]], \mathit{dest}_0) \ ^* \ (j', S', \mathit{dump}', \mathit{dest}')$$

Proof by induction on the length of the computation in $\texttt{pgm}$, using Lemma 1(b) above, if a shortcut is taken by $\texttt{pgm}'$ at the final step. □

We may now draw the desired conclusion:

Theorem 4. *There is a terminating computation in* $\texttt{pgm}$

$$(j_0, [Z]) \ ^* \ (j, S)$$

if and only if there is a terminating computation in $\texttt{pgm}'$

$$(j_0, [Z], [[]], \mathit{dest}_0) \ ^* \ (j, S, \mathit{dump}, \mathit{dest})$$

for some dump and dest.

Proof by Lemma 1(a) and Lemma 3. □

4.2 Linearity

The running time of $\texttt{pgm}'$ is dependent on the flow of control resulting from the inserted shortcuts. We shall use the sequencing structure of pgm as our reference. The terms "loop" and "execution path", in the criteria below, therefore refer to potential flows of control *before* the program is transformed.

Although the actual effect of a $\mathsf{stackop}_{a,b}$ statement is not known until execution time, its influence on the height of the stack(s) may be determined statically: it will increase stack height by $b-a$, or decrease it by $a-b$, depending on whether $a \leq b$ or not. By adding the contributions from each statement it is therefore possible to determine how a particular path through the flow chart from a label j to a label j' will influence the stack height.

As detailed in the theorem below, the following is sufficient to ensure linearity. Any non-empty path from a label j to a label j' which does not decrease stack height must satisfy:

(1) If j labels a $\mathsf{stackop}_{a,b}$ statement with $a > 0$ then the path must contain a shortcut.
(2) If $j = j'$ (*i.e.* if the path is a loop) then the path must contain a shortcut.

As a result of condition (2), there is a limit to how much a path through the flow chart may increase the stack, if the path does not contain a shortcut. Note also that condition (1) disallows $\mathsf{stackop}_{a,b}$ statements with $0 < a \leq b$. On the other hand, if all $\mathsf{stackop}_{a,b}$ statements satisfy $a = 0 \vee a > b$, then the conditions will hold if *all* potential shortcuts are added.

Theorem 5. *Assume that* $\texttt{pgm}'$ *has been constructed in such a way that conditions (1) and (2) are satisfied. Let* C *denote the number of shortcuts, and let* W *be a bound on how much the stack height may increase along any path not containing any shortcut. Then if the computation with* $\texttt{pgm}'$ *terminates, each statement is executed at most* $2 \cdot (C \cdot \#T + 1) \cdot (W + 1)$ *times.*

Proof. Assume $C \geq 1$. The case $C = 0$ is simpler.

By Lemma 1(b), if $\mathit{dest}(j, A) = (j', S')$ then j' labels a $\mathsf{stackop}_{a,b}$ statement with $a > 0$. Each execution of $\mathit{shortcut}(j, A)$ will therefore either enter the pair (j, A) into *dump* or jump to a $\mathsf{stackop}_{a,b}$ statement with $a > 0$. In the first case, let us use the terminology that execution of the shortcut "falls through". Since the computation is finite each pair (j, A), by Lemma 2, will be entered into *dump* at most once. Consequently, a shortcut will fall through at most $C \cdot \#T$ times during the computation.

By condition (1), if a shortcut does not fall through, the computation path leading to the next shortcut, if any, must decrease the height of the stack. Thus the increase in stack height, during the whole computation, cannot exceed the value $(C \cdot \#T + 1) \cdot W$. Taking the initialization with a single element into account one sees that the height of the stack is at most $C \cdot \#T \cdot W + W + 1$.

If a shortcut doesn't fall through, the execution path until the next shortcut (or to a terminal statement) must decrease stack height. This may therefore also happen at most $C \cdot \#T \cdot W + W + 1$ times.

Consequently at most $N = (C \cdot \#T + 1) \cdot (W + 1)$ shortcuts are executed during the computation.

Now consider any particular statement label j. If j is met several times during the computation without any intervening shortcut, then (by condition (2)) stack height is decreased. This may happen at most $C \cdot \#T \cdot W + W + 1 \leq N - 1$ times.

There can be at most $N + 1$ remaining occurrences of j (separated by N shortcuts). The claim of the theorem follows. □

```
S := [Z]; dump := [[]];
initialize dest to the nowhere defined mapping;
i := 0;
while i < n do begin
      i := i + 1;
      push t_i onto S; push [] onto dump
end;
i := 0;
while top(S) ≠ Z do begin
      if top(S) = t_{i+1} then i := i + 1
      else
restore:
      if i > 0 then begin
            if dest(i, top(S)) is undefined then begin
                  add i to the list on top of dump;
                  push t_i onto S; push [] onto dump;
                  i := i − 1;
                  go_to restore
            end;
            i := dest(i, top(S))
      end;
      for each i' in the list on top of dump do dest(i', top(S)) := i;
      pop dump; pop S
end;
write("length of longest palindromic prefix is ", i)
```

Fig. 2. Linear stack program to find for a given string $t_1 t_2 \ldots t_n$ the largest i such that $t_1 t_2 \ldots t_i$ is equal to its own reversal $t_i \ldots t_2 t_1$.

Let us also formulate the result for the frequent simple case where the program deals with one element of the stack at a time. In other words: only stackop statements with arities $(0, 0)$, $(0, 1)$ and $(1, 0)$ occur in pgm.

In this case it is convenient to attach shortcuts precisely to the push statements. The sufficient conditions are simplified into:

(2′) Each loop must contain a push or a pop statement.

Note that a path from a pop to a push statement now automatically leads to a shortcut, and that each loop that does not decrease the height of the stack must contain a shortcut.

Theorem 6. *Assume that* pgm *contains* C $\mathsf{stackop}_{0,1}$ *statements and that the remaining* $\mathsf{stackop}_{a,b}$ *statements have arities* $(0,0)$ *or* $(1,0)$*. Assume furthermore that (2′) is fulfilled and that in the transformation to* pgm′ *shortcuts are attached to all the* $\mathsf{stackop}_{0,1}$ *statements. Then, if the computation with* pgm′ *terminates, in this computation*

- *there will be at most* $C \cdot \#T$ *executions of* push *statements*
- *there will be at most* $C \cdot \#T + 1$ *executions of* pop *statements*
- *each other statement will be executed at most* $2 \cdot (C \cdot \#T + 1)$ *times*

The proof is analogous to that of the general theorem: A push is only executed after a shortcut has fallen through, and that may happen at most $\#T$ times at each particular shortcut. Consequently, no more than $C \cdot \#T + 1$ elements are pushed onto the stack during the computation. This number therefore also bounds the number of executions of a pop statement.

The computation path between any two executions of any other statement must either decrease the stack, which may happen at most $C \cdot \#T + 1$ times, or not decrease the stack, in which case it must contain a shortcut. In the latter case, it must even contain a shortcut that falls through, which may happen at most $C \cdot \#T$ times.

This is the desired result. □

As an additional simplification, shortcuts may be omitted from parts of the program where the logic of the program puts a sufficiently low limit on the number of executions of the statements.

Only the surface configurations actually occurring at shortcuts need be taken into account in the computation of $C \cdot \#T$, and when considering how *dump* and *dest* could be organized. A similar simplification is sometimes also possible with regard to the *range* of *dest* (*e.g.* if there is only one $\mathsf{stackop}_{a,b}$ statement with $a > 0$, the label component is unique and may be omitted).

Figure 2 shows a transformed version of the palindromic prefix algorithm where these improvements have been exploited. No shortcuts have been inserted in the initializing loop because it is obvious that the push is performed only once for each value of i.

Only one push and one pop remain, and they interrupt the remaining loops. The shortcut code is therefore only required at the push, and program labels need not be stored in *dump* or in *dest* (since if a shortcut is taken it will lead from the unique push to the unique pop).

A surface configuration is a value (of i) between 0 and n combined with a stack symbol. The running time (and the size of table *dest*) is thus $\mathcal{O}(n)$.

```
S := [Z];
i := 0;
j := 0;
tryNext:
if i = m then exit(success);
if m − i ≤ n − j then begin
      j := j + 1;
      push 0 onto S;
      go_to tryNext
end;
while j > 0 do begin
      if top(S) = 0 then begin
            pop S;
            i := i + 1;
            if x_i = y_j then begin
                  push 1 onto S;
                  go_to tryNext
            end
      end else pop S;
      i := i − 1;
      j := j − 1
end;
exit(failure)
```

Fig. 3. Naïve algorithm to determine if $x_1x_2 \ldots x_m$ is a subsequence of $y_1y_2 \ldots y_n$

5 Example: Subsequence problem

The m-n-subsequence problem is to determine, for given strings $x = x_1x_2 \ldots x_m$ and $y = y_1y_2 \ldots y_n$, whether x is a subsequence of y in the sense that $x = y_{j_1}y_{j_2} \ldots y_{j_m}$ for some indices $1 \leq j_1 < j_2 < \ldots < j_m \leq n$.

A straight-forward solution procedure would be to generate all the m-combinations $(j_1, j_2, \ldots, j_m)$ and check $\forall i = 1, 2, \ldots, m : \; x_i = y_{j_i}$ for each combination. Generation and checking can be done concurrently, proceeding in order of increasing values of i and backtracking as soon as a mismatch is found, by a program such as the one shown in Figure 3 where the combinations are generated in the natural reverse lexicographic order. (Another way in which this program might have been obtained is indicated in Section 7.) It is not difficult to see that the worst case running time of the program is $\Omega(\binom{n}{m})$. Our transformation will now convert this program to the program in Figure 4 which has the optimal running time[2] $\mathcal{O}(m(n - m + 1))$.

[2] Remark: After more careful thought, it becomes clear that one can also obtain this running time $\mathcal{O}(m(n - m + 1))$ by reprogramming Figure 3 to enumerate the combinations in true lexicographic order. Such insights, however, are nontrivial and not well suited to automation.

```
S := [Z]; dump := [[]];
initialize dest to the nowhere defined mapping;
i := 0;
j := 0;
tryNext:
if i = m then exit(success);
if m − i ≤ n − j then begin
       j := j + 1;
push0: shortcut(push0);
       push 0 onto S; push [] onto dump;
       go_to tryNext
end;
while j > 0 do begin
       if top(S) = 0 then begin
       pop0:  update(pop0);
              pop dump; pop S;
              i := i + 1;
              if x_i = y_j then begin
              push1: shortcut(push1);
                     push 1 onto S; push [] onto dump;
                     go_to tryNext
              end
       end else begin
       pop1:  update(pop1);
              pop dump; pop S
       end;
       i := i − 1;
       j := j − 1
end;
exit(failure)
shortcut(h) ≡ if dest(h, i, j, top(S)) = (h', i', j') then begin
                     i := i'; j := j'; go_to h'
              end else
                     adjoin (h, i, j) to the list on top of dump
update(h)   ≡ for each (h', i', j') on the list on top of dump do
                     dest(h', i', j', top(S)) := (h, i, j)
```

Fig. 4. Improved algorithm to determine if $x_1x_2 \ldots x_m$ is a subsequence of $y_1y_2 \ldots y_n$

The details of the transformation are as follows:

Each loop of the program contains a `push` or a `pop`, and in fact the program contains precisely two `push` statements and two `pop` statements, where one statement in each pair deals with the digit 0 and the other with 1. We introduce four labels, *push0*, *push1*, *pop0* and *pop1*, corresponding to these statements.

Only variables i and j change during computation, within ranges $0 \le i \le m$, $0 \le j \le n$, yielding time bound $m \cdot n$. In fact, the number of different pairs (i, j)

stacked on *dump* and used in *dest* is at most $m(n-m+1)$, since the relation $1 \le i \le j \le n-m+i \le n$ will always hold when *shortcut* is called.

```
      S := [Z];
      push Z onto S;
      push the input pair ⟨n0 over k0⟩ onto S;
argument: shortcut;
      if (let ⟨n over k⟩ = top(S) in 0 < k < n) then begin
            pop ⟨n over k⟩ from S and push ⟨n-1 over k⟩ and ⟨n-1 over k-1⟩ instead;
            go_to argument
      end;
      {0 = k ∨ k = n}
      top(S) := 1;
result:
      swap(S);
      if top(S) is in ℕ then begin
            add;
            go_to result
      end;
      if top(S) is in ℕ × ℕ then go_to argument
      else {top(S) is Z} pop S;
top(S) contains the output
```

Fig. 5. Stack program to compute $\binom{n}{k}$

6 Example: Binomial coefficients

The following recursive definition of the binomial coefficients is valid for $0 \le k \le n$:

$$\binom{n}{k} = \begin{cases} 1 & \text{, if } 0 = k \vee k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{, if } 0 < k < n \end{cases}$$

It is easy to utilize this definition in a stack program. Let the stack alphabet

$$T = \mathbb{N} \times \mathbb{N} + \mathbb{N} + \{Z\}$$

consist of pairs of natural numbers (written in the form $\langle \binom{n}{k} \rangle$, to be used as arguments) and individual natural numbers (used as results) in addition to the bottom marker (Z).

In the simple case ($k = 0$ or $k = n$) the argument pair $\langle \binom{n}{k} \rangle$ is directly replaced by the result (1), but otherwise the top of the stack is replaced by the two subtasks $\langle \binom{n-1}{k} \rangle$ and $\langle \binom{n-1}{k-1} \rangle$.

Whenever the top of the stack exposes a value, the two top elements are swapped. An argument pair will initiate further computation, but if another

value is revealed the two top elements may be added to form a new function result. When swapping uncovers the bottom marker, computation has finished.

The algorithm has been programmed in Figure 5. Initially an additional bottom marker is pushed onto the stack; when this marker reappears, just before termination, it is popped, and the result of the computation is left on the stack.

The program uses some *ad hoc* but hopefully self-explanatory notations for tests and operations. In addition to general pops and pushes there are four general stackoperations S `:=`f`=` S with stack functions f as detailed below:

- `pop` $\langle{n \atop k}\rangle$ `from` S `and push` $\langle{n-1 \atop k}\rangle$ `and` $\langle{n-1 \atop k-1}\rangle$ `instead` corresponds to the f of arity $(0,1)$ where $f(\langle{n \atop k}\rangle) = (\langle{n-1 \atop k-1}\rangle, \langle{n-1 \atop k}\rangle)$
- top(S) `:=` 1 corresponds to the f of arity $(0,0)$ where $f(\langle{n \atop k}\rangle) = (1)$
- `swap(`S`)` corresponds to the f of arity $(1,1)$ where $f(v_0, v_1) = (v_1, v_0)$
- `add` corresponds to the f of arity $(1,0)$ where $f(v_0, v_1) = v_0 + v_1$

The running time of the program is $\mathcal{O}(\binom{n}{k})$, as is easily seen, and the main reason for this behaviour is the long-winded recomputation of many partial results.

The program may, however, be subjected to the transformation of Section 3.1 by inserting a shortcut at the label *argument*, as in Figure 5. Since the pair $\langle{n \atop k}\rangle$ in the top of the stack at this point will always satisfy $0 \le k \le k_0$ and $0 \le n - k \le n_0 - k_0$, *dest* only needs $(n-k+1)(k+1)$ entries.

The transformed program in effect implements the method of "Pascal's triangle", using $(n-k)k$ additions to compute $\binom{n}{k}$. If more complex operations such as multiplications were allowed, faster methods could be devised.

Linearity (in the number of surface configurations) does not quite follow from Section 4.2, since condition (2) is satisfied but condition (1) is violated by the path from "*result*" via a negative outcome of the test "top(S) is in $\mathbb{N}$" and positive outcome of "top(S) is in $\mathbb{N} \times \mathbb{N}$" to *argument*, which does not decrease stack height and also does not contain a shortcut.

A more detailed analysis of the actual case reveals that our transformed program is still linear: Of the two argument pairs that are created by "`pop` $\langle{n \atop k}\rangle$ `from` S `and push` ... `instead`" one is treated immediately by the next execution of the *argument*-loop; the other is eventually brought to the top of the stack when being swapped with the result value of the first pair. The offending path is therefore taken the same number of times as the number of executions of "`pop` $\langle{n \atop k}\rangle$ `from` S `and push` ... `instead`" ($(n-k)k$ times).

6.1 Generalization

The structure used in Figure 5 may be adjusted to compute any function f with a recursive definition

$$f(x) = \begin{cases} c_1 & \text{, if } p_1(x) \\ \vdots & \vdots \\ c_q & \text{, if } p_q(x) \\ f(d_1(x), \ldots, d_r(x)) & \text{, otherwise} \end{cases} \qquad (*)$$

where $d_1, \ldots, d_r$ are decreasing functions in some well-founded ordering of the argument domain. Instead of swap a circular rotation of the topmost r elements of the stack could be used.

Our method may in this way be said to reinvent "course-of-values recursion" or "dynamic programming" for definitions of the form (*).

7 Conclusion

It has been shown how any stack program in a mechanical way may be transformed into a version that uses some extra tables (whose size is determined by the number of surface configurations), but such that the execution time of the transformed version is proportional to the number of surface configurations.

For this insight to be useful for a general computational problem one should first solve the problem by a stack program with a small number of surface configurations and then apply the transformation. In many cases, the first of these stages may require ingenuity; for example, to obtain the stack program for pattern matching from a naïve version using two pointers does not seem obvious.

An interesting source of stack programs arises as results of Floyd's transformation (in [6]) of non-deterministic programs to deterministic ones. These are obtained by "running the non-deterministic program in reverse", so to speak, using a stack to take care of the bookkeeping involved in backtracking.

```
i := 0;
j := 0;
while i < m do begin {x_1 ... x_i is a subsequence of y_1 ... y_j}
      if m − i > n − j then failure;
      j := j + 1;
      case choose(2) of
      0:    skip;
      1:    begin
                  i := i + 1;
                  if x_i ≠ y_j then failure
            end
      end case
end;
success
```

Fig. 6. Non-deterministic algorithm to determine whether $x_1 x_2 \ldots x_m$ is a subsequence of $y_1 y_2 \ldots y_n$

A particularly nice example of this procedure (suggested to us by Torben Mogensen[9]) is the non-deterministic program for the subsequence problem shown in Figure 6. This program is a straightforward product of the problem

specification, but resolving non-determinism by systematically trying case 0 first and case 1 afterwards gives the deterministic program on Figure 3 which may be subjected to Cook's transformation, resulting in the program with optimal running time shown in Figure 4. (If the possibilities are examined in the order 1 first, then 0, the optimal program is produced directly.)

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company 1974.
2. Torben Amtoft Hansen, Thomas Nikolajsen, Jesper Larsson Träff, and Neil D. Jones, Experiments with Implementation of two Theoretical Constructions, p. 119–133 in Logic at Botik, *Lecture Notes in Computer Science* Vol. 363, Springer-Verlag 1989.
3. Richard S. Bird, Improving Programs by the Introduction of Recursion, *Communications of the ACM* Vol. 20 No. 11 (November 1977) 856–863.
4. Stephen A. Cook, Linear-Time Simulation of Deterministic Two-Way Pushdown Automata, p. 75–80 in C. V. Freiman (editor): *Information Processing 71*, North-Holland Publishing Company 1972.
5. Stephen A. Cook, Characterization of Pushdown Machines in Terms of Time-Bounded Computers, *Journal of the ACM* Vol. 18 No. 1 (January 1971) 4–18.
6. Robert W Floyd, Nondeterministic Algorithms, *Journal of the ACM* Vol. 14 No. 4 (October 1967) 636–644.
7. Neil D. Jones, A Note on Linear Time Simulation of Deterministic Two-Way Pushdown Automata, *Information Processing Letters* Vol. 6 No. 4 (1977) 110–112.
8. Neil D. Jones, Carsten Krogh Gomard, Peter Sestoft: *Partial Evaluation and Automatic Program Generation*, Prentice Hall International, 1993.
9. Torben Ægidius Mogensen: Personal communication, September 1993.

With the exception of an errorcorrection this is a copy of the paper (same title) page 1–18 in Juhani Karhumäki, Hermann Maurer, and Grzegorz Rozenberg (eds.): Results and Trends in Theoretical Computer Science, Proceedings of a Colloquium in Honor of Arto Salomaa, Graz, Austria, June 1994, Lecture Notes in Computer Science volume 812, Springer-Verlag.