ma
the
ma
tisch

cen
trum

AFDELING INFORMATICA IW 48/75 SEPTEMBER

L. AMMERAAL

EXTENDING A RUN-TIME STACK WITH SOME REGISTERS

Prepublication

amsterdam 1975

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 48/75 SEPTEMBER

L. AMMERAAL

EXTENDING A RUN-TIME STACK WITH SOME REGISTERS

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 68L15

ACM -Computing Reviews- category: 4.12

EXTENDING A RUN-TIME STACK WITH SOME REGISTERS

by

L. Ammeraal

ABSTRACT

This paper describes how a variable number of registers (or accumulators) can be used as the upper part of a virtual stack. It formally defines the concept of a virtual stack and outlines how a compiler for an ALGOL-like language can generate instructions for such a stack.

KEY WORDS & PHRASES: *stack, registers, compiler, code generation, ALGOL 68*

INTRODUCTION

Many conventional machines have fast register-to-register instructions. Programmers are often recommended to avoid unnecessary load and store instructions because these are more expensive. For the implementer of high-level languages such an advice does not conform to his need for elegant and generally applicable addressing methods. Registers (or accumulators, as they are sometimes called) are usually available in a very limited number. They often have the awkward property to be only statically addressable, i.e. one can use, e.g., register R_5 but not R_i where i is not known before execution time. It has sometimes been said that a machine should either have no registers, or infinitely many. Ignoring the advice about register exploitation and performing all operations on a run-time stack is possible but nót satisfactory for machines like the Control Data 6600. A first step to improve this is having one register available as the top of the stack. This means that memory access is not actually done in the following two situations:

a. a value is to be pushed to the stack and the register is "free",

b. a value is to be popped from the stack and the register is "occupied".

A boolean variable *stock* can be used at compile-time to indicate whether the register is to be considered occupied or free; the reduction in stack operations can be visualized by the deletion of all sequences "PUSH; POP" in the object program. My first paper [1] on the Mini ALGOL 68 compiler describes this in more detail. With only one register as topmost stack element, still much memory access is necessary. A natural attempt to reduce this is to generalize the idea mentioned above with respect to the number of registers. To keep this presentation as simple as possible I chose to use at most three registers as top stack elements. At any moment a variable number of these three registers are actually occupied. In a more general discussion we would have two variable numbers of registers, of which one is more variable than the other. Fixing the maximum number to three is less confusing and this number can easily be replaced by a different one, if wanted. The ideas presented in this paper were successfully implemented in the current version of the Mini ALGOL 68 compiler.

TERMINOLOGY

A <u>proper</u> <u>stack</u> S is an n-tuple $(S_1,S_2,\ldots,S_n)$, where $S_1,S_2,\ldots,S_n$ are contiguous storage cells. The number n is called the <u>(proper)</u> <u>height</u> of S and is to be considered variable ($n \geq 0$). Storage cell S_n is called the <u>(proper)</u> <u>top</u> of S. We will say that a value x is <u>properly</u> <u>pushed</u> if the sequence of statements

$$n := n+1;\ S_n := x$$

is executed.

Similarly, the value of S_n is <u>properly</u> <u>popped</u> to x by

$$x := S_n;\ n := n-1.$$

A register configuration is an m-tuple $(R_{p_1},R_{p_2},\ldots,R_{p_m})$, where

$m \in \{0,1,2,3\}$ (as motivated in the introduction),

$p_i \in \{1,2,3\}$ $(i=1,\ldots,m)$, and

$p_{i+1} \equiv p_i+1 \pmod 3$ $(i=1,\ldots,m-1)$.

We thus have the following ten register configurations:

$\emptyset$	(m=0)
(R_1), (R_2), (R_3),	(m=1)
(R_3,R_1), (R_1,R_2), (R_2,R_3),	(m=2)
(R_2,R_3,R_1), (R_3,R_1,R_2), (R_1,R_2,R_3),	(m=3).

A <u>virtual</u> <u>stack</u> S' is an (n+m)-tuple

$$(S_1,S_2,\ldots,S_n,R_{p_1},R_{p_2},\ldots,R_{p_m})$$

which is composed of the proper stack $(S_1,S_2,\ldots,S_n)$ and the register configuration $(R_{p_1},R_{p_2},\ldots,R_{p_m})$. The *(virtual) height* of S' is $n+m$.

In the following we shall use t for p_m. The *(virtual) top* of S' is R_t if $m > 0$, or S_n if $m=0$. We now observe that each triple (n,m,t), where

$$n \in \{0,1,2,\ldots\}$$

$$m \in \{0,1,2,3\}$$

$$t \in \{1,2,3\},$$

uniquely determines a virtual stack

$$S' = (S_1,S_2,\ldots,S_n,R_{p_1},\ldots,R_t)$$

(The reverse does not hold if $m=0$).
This can easily be verified from the four lines with register configurations listed above.
If $m = 0$, the register configuration is $\emptyset$.
If $m > 0$, the register configuration is the t-th one on the m-th line of the last three of those lines.

SOME USEFUL CODE GENERATING AND BOOKKEEPING PROCEDURES

It will now be shown how a value is brought to or taken from the virtual stack. For reasons of brevity I will use (Mini) ALGOL 68 for this purpose. Those who are unfamiliar with this language should first pay some attention to the following examples, explained by ALGOL 60.

```
Mini ALGOL 68              | ALGOL 60
                           |
a)  (a < b | c | d);       | if a < b then c else d;
                           |
b)  proc f = (real a) int: | integer procedure f(a); value a; real a;
    (s;b);                 | begin s; f:= b
                           | end;
```

c)	*proc p = (int a) void: S;*	*procedure p(a); value a; integer a; S;*
d)	*n+:= 1*	*n:= n + 1*

After this explanation the following very simple functions *newer* and *older* will now immediately be clear. They yield the cyclic successor and predecessor of a given element *k* in the triple *(1,2,3)*.

```
proc newer = (int k) int : (k=3 | 1 | k+1);
proc older = (int k) int : (k=1 | 3 | k-1);
```

The procedures of this section are part of a compiler. It is now interesting to notice that *m* and *t* are known at compile-time, whereas the proper height *n* is not. Increasing and decreasing *n* is not done by the compiler but at run-time. It is the task of the compiler to generate instructions for these and other operations. These instructions are here generated by means of the output procedure *out*. Because *m* and *t* are compile-time variables, only their values and not the variables themselves must be output. Therefore the following notation is adopted here for the actual parameters of *out*. A string between quote symbols *(")* is actually output, but something of the form

$$\{\ .....\ \}_x$$

must first be transformed into a string by replacing all occurrences of x inside the braces by the value of x. So if, e.g., $t=2$ then the instruction

$$S_n := R_2$$

is written by

$$out\ (\{S_n := R_t\}_t).$$

The following procedure changes the global variables m and t such that R_t will become available to put something in.

```
proc newtop = void:
(t := (m=0 | 1 | newer (t));
   (m < 3 | m+:= 1   | out ("n+:=1"); out ($\{S_n :=R_t\}_t$))
);
```

At first sight it may seem wrong to (properly) push R_t. It is correct, however, because if this happens then $m=3$ and, immediately before, t :=*newer*(t) was executed, which means that R_t is now in fact the "oldest" register. Let us assume, e.g., the virtual stack to be

$$(S_1, S_2, S_3, S_4, R_3, R_1, R_2).$$

Then $(n, m, t) = (4, 3, 2)$. The effect of *newtop* is now

$$t := 3;\ n := 5;\ S_5 := R_3,$$

yielding the new virtual stack

$$(S_1, S_2, S_3, S_4, S_5, R_1, R_2, R_3)$$

of which R_3 has still to be filled.

The following two procedures ensure that, after the call, $m \geq 1$ and $m \geq 2$, respectively.

```
proc atleast1reg= void:
(m=0 | out ("$R_1$ := $S_n$"); out ("n-:=1"); m:=1; t:=1);
proc atleast2reg = void:
(m < 2
| (m=1
  | int t1 := older (t); out ($\{R_{t_1} := S_n\}_{t_1}$); out ("n-:=1")
  | out ("$R_2$ := $S_n$"); out ("$R_1$ := $S_{n-1}$"); out ("n-:=2"); t :=2
  );
  m :=2
);
```

AN EXAMPLE.

Even with only three registers the number of proper stack operations is reduced considerably in practice. In some simple but frequently occurring situations the proper stack is not used at all. As an example, consider the assignation

x := yy+1,

where the modes of *x* and *yy* are specified by <u>*ref*</u> <u>*int*</u> and <u>*ref*</u> <u>*ref*</u> <u>*int*</u>. (This example was also dealt with in [1], using only one register). We assume that, initially, *m=0*. The example can be written in Reverse Polish as

x, yy, deref, deref, 1, add, assign.

The elaboration of *x* and *yy* give rise to calls of *newtop*. Because we assume *m=0* as initial state, these calls yield R_1 and R_2. The meaning of *deref* is dereferencing, which can be implemented by replacing an address by the contents of this address.
In our case the address is in the virtual top R_2. In general *atleast1reg* is called to ensure that $m \geq 1$ before dereferencing. Both *add* and *assign* can be considered as dyadic operations. Therefore *atleast2reg* is called by each of them, but here, too, the required number of registers is already available, so no run-time actions are involved. Dyadic operations decrease the height of the virtual stack by one. They are implemented as follows:

```
atleast2reg; t1 :=t; t :=older(t); m-:=1;
out ({$R_t$ := $R_t$ op $R_{t_1}$}$_{t,t_1}$).
```

Thus the object code will be:

```
R1 := x;
R2 := yy;
R2 := contents of the address given in R2;
R2 := contents of the address given in R2;
```

$R3 := 1;$

$R_2 := R_2 + R_3;$

$S_{R_1} := R_2.$

As the last instruction shows, identifiers like x are associated with certain elements of the proper stack S. If S_i is this element for x then R_1 contains i, the address of x. This address is the result of the given assignation: an assignation (a priori) yields a value in (Mini) ALGOL 68. In the compiler, $m=1$ and $t=1$ after the generation of these instructions. If the result of the assignation has to be voided, e.g. if it is followed by a semicolon, then the following procedure is called.

```
proc voiding = void:
( m=0 | out ("n-:=1") | m-:=1;
                         (m > 0 | t := older (t))
).
```

In our case this will set m to 0 and produce no object code.

THE VIRTUAL STACK AT RUN-TIME ROUTINE ENTRY AND EXIT.

If a procedure is called at run-time, care must be taken that the routine uses the virtual stack with initial values of m and t as they were at the moment of the call.

A similar condition must be satisfied on routine exit. As mentioned before, m and t are compile-time quantities. They may have different values at the various points where a given procedure is called. Therefore some kind of normalization is necessary before and after the execution of a routine. In the implementation of Mini ALGOL 68 the values of actual parameters must reside on the proper stack instead of in registers. This is so because these values are accessed through their corresponding formal parameters and these are, like normally declared identifiers, identified by their position in the proper stack S', as mentioned in the last example. Therefore the contents of $R_{p_1},\ldots,R_t$ are, in this order, properly stacked before the call is carried out and $m=0$ is assumed at the beginning of the routine. On routine exit normalizing to $m=0$ is done if no value is delivered, i.e. for a procedure yielding *void*. For other procedures we normalize to $m=1$, $t=1$.

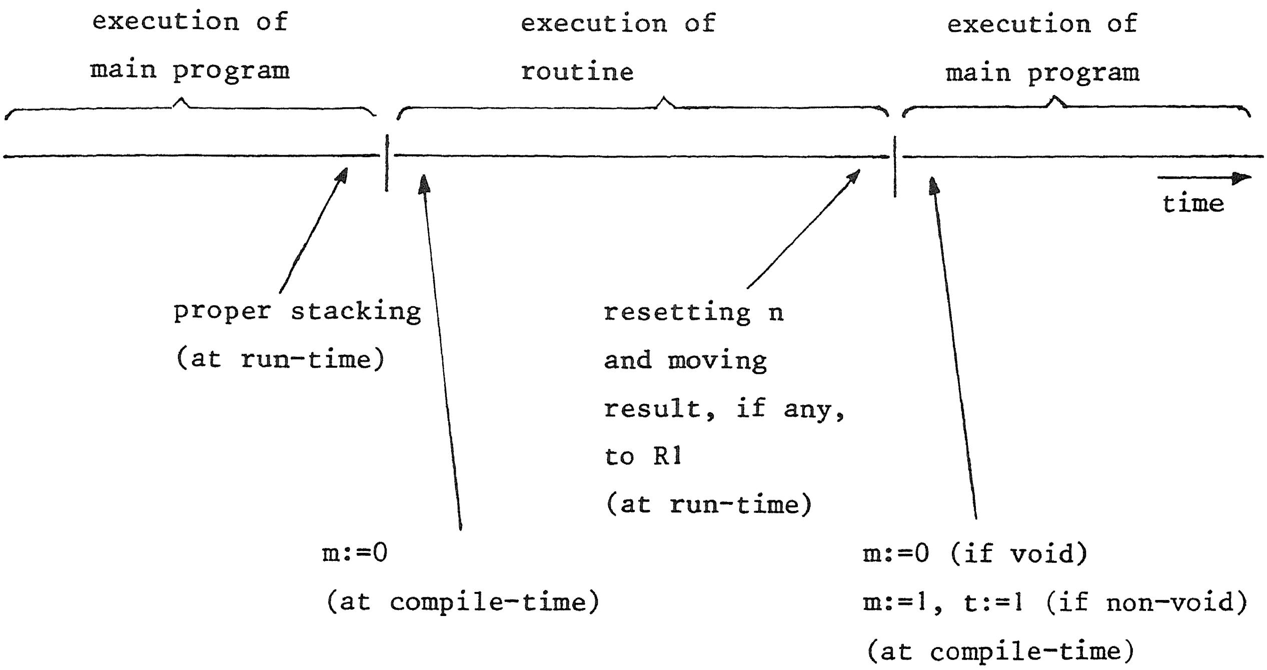
This is accomplished in the compiler as outlined below.

```
proc routinetext = ...
(int mold :=m; told :=t; m :=0;
     now the syntactic procedure unit
     is called which generates code
     for the procedure body;

if void is yielded
then output instructions for resetting n
     to the value at call time
else output the same instructions, but,
     additionally put the result which
     was at the top of the old virtual
     stack into R1
fi; output the return jump;
m := mold; t :=told
);

proc call = ...
( ...
  output instructions to (properly) stack $R_{p_1},\dots,R_t$;
  output the jump to the routine and
  output the return label
  if void is yielded then m :=0 else m :=1; t :=1 fi
)
```

These compile-time and run-time modifications on the virtual stack can be depicted on a time axis as follows.


execution of main program
execution of routine
execution of main program
time
proper stacking (at run-time)
resetting n and moving result, if any, to R1 (at run-time)
m:=0 (at compile-time)
m:=0 (if void)
m:=1, t:=1 (if non-void)
(at compile-time)

CONCLUSION

The concept of a virtual stack implies two levels of abstraction. At the most abstract level we consider values simply to be pushed to and popped from a stack. At a less abstract level we are aware that this stack is virtual and comprises a proper stack extended with some registers. We are thus enabled to combine straightforward stack-oriented translation methods with a good use of registers.

REFERENCES

[1] AMMERAAL, L., *An implementation of an ALGOL 68 sublanguage,* Proceedings of the International Computing Symposium 1975, North-Holland Publishing Company, Amsterdam (1975), pp. 49-53.

[2] AMMERAAL, L., *Mini ALGOL 68 User's Guide,* Mathematical Centre IW 32/75, Amsterdam (1975).