Display

CLM-P 270

CLM-P 270

L

United Kingdom Atomic Energy Authority

RESEARCH GROUP

Preprint

THE TRANSLATION OF SYMBOLIC ALGOL I TO SYMBOLIC ALGOL II BY THE STAGE2 MACRO-PROCESSOR

G. KUO-PETRAVIC
M. PETRAVIC
K. V. ROBERTS

Culham Laboratory
Abingdon Berkshire

1970

CLM - P 270

THE TRANSLATION OF SYMBOLIC ALGOL I TO SYMBOLIC ALGOL II BY THE STAGE2 MACRO-PROCESSOR

by

G. Kuo-Petravic*
M. Petravic*
K.V. Roberts

(Submitted for publication in Journal of Computational Physics)

ABSTRACT

This paper describes briefly the facilities offered by the Stage2 macro-processor itself. The logic of the translator program written in Stage2 (TRANSTAG) is then described fully using as an example the magnetic equation in the Trinity program. The reduction to reverse Polish and the subsequent decoding back to a set of nested procedures in Symbolic Algol II are fully explained.

* Dept. of Engineering, University of Oxford

U.K.A.E.A. Research Group,
Culham Laboratory,
Abingdon, Berks.

November, 1970

As has been noted in a previous paper(1), Symbolic Algol I has a form closely analogous to that of mathematical physics and can therefore serve as a language of communication between computational physicists. This should lead to a rapid progress in the interchange of ideas and programs which have so far been hampered to some degree by the machine dependence of all programs and the lack of a sufficiently flexible method of communication. Symbolic Algol I, although being legal Algol, uses the language in such a way that the actual physics contained in a program can be expressed in a concise, intelligible and general form, in much the same way as in mathematical physics. However, by its very generality, in order that the detailed operation of an operator is only contained in the actual procedure called after the name of the operator, it has lost large factors in speed because of the necessity of repeated procedure calls. Therefore, in order that Symbolic Algol I be a useful language, some automatic technique of optimization must be developed. Furthermore the method should be a relatively simple one such that it is easy to implement on any machine with its existing compilers and peripheral software. In this series of papers we set out to show how a few orders of magnitude gain in speed may be achieved by a two stage process of optimization involving first the use of the Stage2 macro-processor and then Algol itself. In this paper we shall concern ourselves with the first stage of the optimization, which is the translation of Symbolic Algol I into Symbolic Algol II by the Stage2 macro-processor.

The Stage2 macro-processor is based on a language called Flub developed principally by W. Waite(2) at the University of Colorado. It uses a method of macro template matching which is very convenient for the manipulation of operators and operands in a mathematical equation.

The entire translator program which converts Symbolic Algol I to Symbolic Algol II is written as a series of Stage2 macros. This program, named TRANSTAG, is about 500 card images long, and because of its length it will not be reproduced in its entirety here. Readers who are interested in using it can obtain a report(3) and paper tape from the authors. In this section we shall briefly describe the logic behind the method used to translate the Symbolic Algol I input into a set of nested procedures calls, which can then act as input to the optimizer program written in Algol.

The text to be analysed is concatenated to the end of TRANSTAG and the program will scan every line and try to match it with a macro template with variables in specified 'slots'. If no match can be found, the line is output without alteration. Suppose the line in the text reads: Example x = y + z . , this could be matched by a macro: Example '=' . , where ' stands for a parameter flag, and . the macro end of line flag. In this case parameter 1 will consist of x and parameter 2 of y + z. Within the macro, operations can be performed on the two parameters 1 and 2 and any text containing some function of the parameters may be output if required. The basic use of Stage2 lies in its facility to translate instructions on certain parameters in one language to the same or different instructions on the same set of parameters in another language. It may almost be said that it is machine independent in that it is easily implemented on any machine, taking at most one week. Apart from the usual facilities of conditional jumps, skips, do-loops etc., there is one facility in Stage 2 which is particularly useful for text manipulation. An example of this is the following:

```
' 10 ' 17 + - * /.

- - - - -

- - - - -

' F8.
```

Inside a macro, an exact copy of parameter 1 is written as '10. The do-loop shown above means act on parameter 1 using + - * or / as break characters up to 'F8 and repeat again, so that inside the loop only a fraction of the original parameter is considered as the current parameter 1. If parameter 1 consisted of X + YZ *ABC, then the first time round X would be the current parameter 1 and next time round YZ and so on. There are a few more values which are easily evoked: (i) '|| means the content of parameter 1, here we have the very useful facility that any character, symbol, or word can be the address or content of any other symbol or word, (ii) '13 means an exact copy of the break character following the current parameter 1, (iii) '15 gives the length in characters of the current parameter 1.

We shall now proceed to illustrate our method by way of an example which is the RHS of the magnetic equation used in the Trinity program(4):

```
B = dT * (Curl ( Cross (v,B)) + eta * Delsq (B));
```

The first step is to replace the operator names by symbols such as α, β, γ etc. We note that the operators are of two kinds; the function operator like Curl and Delsq which operates on its RHS only, and the infix operator like Cross which acts between two operands. The latter is written in Symbolic Algol I as a procedure with two arguments. It is desirable at this stage to replace Cross (v,B) by ((v)α(B)), where α is any symbol, to bring it in line with other normal operators e.g. */. This is achieved with a macro Recode ','. which separates the statement into two halves. For the sake of illustration, let us take here a more complex example:

```
- - - - * Cross ( v + Curl (B), Curl ( Cross (v,B))) + - - - -
- - - - * Cross < v + Curl (B), Curl ( Cross (v,B))> + - - - -
- - - - *        ((v + Curl (B))α(Curl (Cross (v,B)))) + - - - -
```

We set up a counter which counts brackets, taking opening bracket (= +1 and closing bracket) = −1. Taking first the second half of the line, we count to the first unbalanced closing bracket and replace it by $\rangle$. Similarly the first unbalanced opening bracket counting backwards in the first half of the line is replaced by $\langle$. The name Cross is coded into a symbol such as α and the following substitution made:

$$\langle \rightarrow (\; ($$
$$, \rightarrow) \; \alpha \; ($$
$$\rangle \rightarrow) \;)$$

Returning to the case of the magnetic equation, we would then have:

```
B + dT * ( Curl ((((v) α (B))) + eta * Delsq (B))
```

Function operator names like Curl and Delsq are then coded. They are recognised by the fact that they appear before an opening bracket (and after an operator or another opening bracket (. We then have

```
B + dT * ( β (((v) α (B))) + eta * γ (B))
```

where Curl has been coded into β and Delsq into γ. In order to take into account the operator precedence implied in this statement by the configuration of its brackets and the nature of its operators, this statement is turned into a one dimensional reverse polish string by the use of about ten short macros. The rules for output in reverse polish are: (i) Operands are written into the reverse polish string in a backward sequence. A dummy operand denoted by Ω is written to the string when a function operator appears. (ii) Operators are stacked on an operator stack. If the operator on top of the stack is of higher or equal priority to the incoming operator, then the operator on stack is output. (iii) Brackets are treated as special operators. An opening bracket (goes on stack without comparison, and a closing bracket) causes the output of all operators down to the previous opening bracket (and removes this opening bracket (from stack.

The resultant reverse polish string reads:

+*+*γB Ω eta βαB v Ω DT B

where a blank space immediately following an operand is inserted to act as break character for operands.

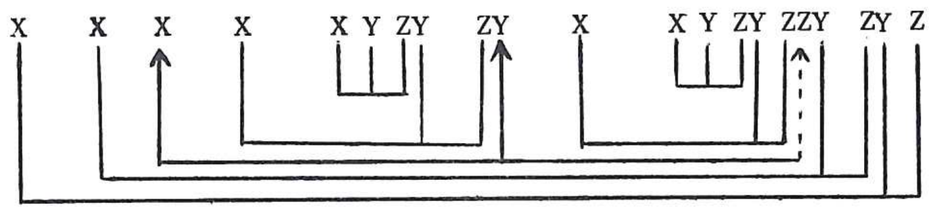
As the operator symbols are the addresses for the operator names, it is easy by invoking '11 to retrieve the operator names. However, some rules regarding the output of brackets and commas have to be formulated. The reverse polish string is analysed by a loop using the operator symbols and blank space as break characters. The operands are recognized by the fact that the current length of parameter 1 is non zero, whereas for operators the current parameter 1 is the null string. The output format is determined by the following rules:

	output	add to test string
operator preceded by operator	operator (	X
operator preceded by operand	,operator (	YX
operand preceded by operator	operand	
operand preced by operand	,operand)	YZ

The configuration of brackets and commas built up in this process must be remembered in order that correct sets of opening bracket, comma, and closing bracket —— (,) should be present when the output is completed.This is achieved by adding to a test string the letter X whenever and opening bracket —— (is output, Y whenever a comma —— , is output, and Z whenever a closing bracket ——) is output. In our example we have:

+*+*γB Ω eta βαB v Ω dT B

sum(mult(sum(mult(Delsq(B,Ω),eta),Curl(Cross(B,v),Ω)),dT),B)



Whenever a Z is added to the test string, a triad of XYZ is completed and can be removed from the string. We then tentatively assign another closing bracket, that is to say, add another Z to the remainder of the test string and test if a sequence of XYZ exists at its end. If yes, a closing braket) can be definitely output at this point and another XYZ deleted from the string. When no more sequence of XYZ could be found at the end of the test string, the last Z is removed. It may be seen from the example that the closing bracket in dotted line has been assigned in this way. If this were absent, there would then be an unbalance of brackets and commas when the end of the line is reached. At this point the test string should become a null string.

As a consequence of the use of reverse polish, all operands appear in reverse order to that in the original statement. While this presents no difficulty to the arithmetic unit of a computer, it is undesirable in Symbolic Algol II as we do not want to define operators like Curl in different order from their conventional definition. We, therefore, perform an interchange of operands within each triad of opening bracket, comma and closing bracket —— (,). This is easily achieved by a macro called Interchange ' . . A counter which counts opening bracket (= +1, comma , = 0, and closing bracket) = -1 enables us to pick out the appropriate triad:

```
counter   1   2   3    4      5 5 4     33     4       5 5 44 322  1  0
         sum(mult(sum(mult(Delsq(B,Ω),eta),Curl(Cross(B,v),Ω)),dT),B)
         sum/mult(sum(mult(Delsq(B,Ω),eta),Curl(Cross(B,v),Ω)),dT)$/B\
         sum⟨B:mult(sum(mult(Delsq(B,Ω),eta),Curl(Cross(B,v),Ω)),dT)⟩
```

When the counter registers 1 and the break character is an opening bracket —— (, this is replaced by /. When the counter again registers 1 and the break characer is a comma —— , , this is replaced by $/. Finally when the counter egisters 0 and the break character is a closing bracket ——), this is replaced by \. We then simply subject this line to a macro Swop '/' $/ '\ in which a line given by '10⟨'30:'20⟩ is created and stored. Operands within this triad have then been interchanged and the corresponding brackets inactivated. This line is re-submitted to the macro Interchange '. , and this time the next level down will be interchanged. This continues until we obtain:

```
sum⟨B:mult⟨DT:sum⟨Curl⟨Ω:Cross⟨v:B⟩:mult⟨eta:Delsq⟨Ω:B⟩⟩⟩⟩⟩.
```

Finally by replacing ⟨ → (, : → , , and ⟩ →) , and deleting the dummy operand Ω , we obtain the desired result:

```
Equate (B,sum(B,mult(DT,sum(Curl(Cross(v,B)),mult(eta,Delsq(B)))))).
```

The procedure Equate has been created separately by a macro which matches the input equation.

In Figure 1a, we list the input to TRANSTAG as obtained from a Symbolic Algol I version of a 3-dimensional MHD programme called Trinity(4). In Figure 1c, we list the corresponding output. However, with a trivial modification, TRANSTAG can be made to handle input as shown in Figure 1b where the equations are written in an entirely mathematical form. Here the user has to define <x> ≡ Cross, <D> ≡ dot etc., although in principle symbols such as ∇x, ∇^2 may be created on the teletype thus making the definitions unnecessary.

On the KDF9 at Culham Laboratory, TRANSTAG took 18 minutes to process the four equations of Trinity. This is not extremely fast; however, such translation need only be performed once for a given problem and thus this time would be quite tolerable.

Acknowledgements

The authors wish to express their thanks to Mr W.Holman for assistance in Stage2 programming, and to Dr P.C. Poole for the implementation of Stage2 at Culham and for many helpful discussions.

References

(1) M. Petravic, K.V. Roberts and G. Kuo-Petravic, 'The automatic optimization of Symbolic Algol programmes', to be submitted to the Journal of Computational Physics.

(2) W.M. Waite, 'The Mobile Programming System STAGE2', Comm. ACM 13 415, 1970.

(3) G. Kuo-Petravic, M. Petravic, K.V. Roberts. 'The translation of Symbolic Algol I to Symbolic Algol II by the Stage2 macro-processor, A User's guide' PDN 4/71. Obtainable from Culham Laboratory.

(4) K.V. Roberts and J.P. Boris, 'The Solution of partial differential equations using a Symbolic Style of Algol', CLM-P 261, 1971.

```
M NEW.RHO=RHO-DT*DIV(RHO*V);
 'FOR' C1=1 'STEP' 1 'UNTIL' 3 'DO'
M V=(RHO*V+DT*(-GRAD(RHO*TEM+DOT(B,B)/2)-DIV(RHO*TEN(V,V)-TEN(B,B))
M    +NU*DELSQ(RHO*V)))/NEWRHO;
 'FOR' C1=1 'STEP' 1 'UNTIL' 3 'DO'
M B=B+DT*(CURL(CROSS(V,B))+ETA*DELSQ(B));
M TEM=TEM+DT*(-DIV(TEM*V)+KAPPA*DELSQ(TEM)
M      +(2-GAMMA)*SAV(TEM)*DIV(V)+(GAMMA-1)*ETA*DOT(CURL(B),CURL(B))
M      /SAV(RHO)+(GAMMA-1)*NU*(DOT(CURL(V),CURL(V))+DIV(V)↑2));
```

(a)

```
M  SCALAR EQUATION 1;
M  DRHO/DT =-DIV(RHO*V);
M  VECTOR EQUATION 2;
M  DV/DT=(DIV((B<T>B)-RHO*(V<T>V))-GRAD(RHO*TEM+(B<D>B)/2.U)
M          +NU*DELSQ(RHO*V))/RHO;
M  VECTOR EQUATION 3;
M  DB/DT=CURL(V<X>B)+ETA*DELSQ(B);
M  SCALAR EQUATION 4;
M  DTEM/DT=-DIV(TEM*V)+KAPPA*DELSQ(TEM)
M             +(2-GAMMA)*SAV(TEM)*DIV(V)+(GAMMA-1)*ETA*(CURL(B)<D>CURL(B))
M             /SAV(RHO)+(GAMMA-1)*NU*((CURL(V)<D>CURL(V))+DIV(V)↑2));
```

(b)

```
EQUATE( NEW RHO,DIFF(RHO,MULT(DT,DIV(MULT(RHO,V)))))-(000600)
'FOR' C1=1 'STEP' 1 'UNTIL' 3 'DO'-(000700)
EQUATE( V,QUOT(SUM(MULT(RHO,V),MULT(DT,SUM(DIFF(DIFF(BLANK,GRAD(SUM(MUL
  (004200)
T(RHO,TEM),QUOT(DOT(B,B),RNUM(2))))),DIV(DIFF(MULT(RHO,TEN(V,V)),TEN(B,
  (004300)
B)))),MULT(NU,DELSQ(MULT(RHO,V)))))),NEWRHO))-(004400)
'FOR' C1=1 'STEP' 1 'UNTIL' 3 'DO'-(004500)
EQUATE( B,SUM(B,MULT(DT,SUM(CURL(CROSS(V,B)),MULT(ETA,DELSQ(B))))))-(00
  5600)
EQUATE( TEM,SUM(TEM,MULT(DT,SUM(SUM(SUM(SUM(DIFF(BLANK,DIV(MULT(TEM,V))
  (009800)
),MULT(KAPPA,DELSQ(TEM))),MULT(MULT(DIFF(RNUM(2),GAMMA),SAV(TEM)),DIV(V
  (009900)
))),QUOT(MULT(MULT(DIFF(GAMMA,RNUM(1)),ETA),DOT(CURL(B),CURL(B))),SAV(R
  (010000)
HO))),MULT(MULT(DIFF(GAMMA,RNUM(1)),NU),SUM(DOT(CURL(V),CURL(V)),EXP(DI
  (010100)
V(V),2)))))))-(010200)
```

(c)

Fig. 1