BIT 5 (1965), 151–163

CHECKING OF OPERAND TYPES IN ALGOL COMPILERS*

PETER NAUR

Abstract.

The paper describes the method used in the GIER ALGOL compiler for checking the source program for the compatibility of the kinds and types of identifiers and constants as given in declarations with the use of these entities as operands in the statements and expressions of the program. The basic method is a pseudoevaluation of the expressions of the program. This proceeds like a run-time evaluation, but works with descriptions of the types and kinds of the operands instead of with values. Several examples of the use of this method are given. A general way of achieving suppression of redundant error messages while still retaining the power to continue the checking process when an error has been detected is also described. Further, a classification of the operands of ALGOL 60 for use in the process is given. The paper ends with a sample error report produced by the GIER ALGOL compiler, illustrating the power of the method.

Introduction.

The practical value of a compiler for a problem oriented language on an electronic computer depends greatly on its facilities for detecting programming errors. Programming errors are in fact so frequent that powerful methods of error detection may be expected to reduce the average programmer and machine time used for check-out of new programs by factors of between 2 and 5. For this reason the problem of error detection received a high priority when the GIER ALGOL compiler (ref. 1) was designed.

The basic philosophy behind the treatment of error in GIER ALGOL is the following: (1) No program in which the compiler has detected a formal error will be permitted to run. (2) When an error has been detected and reported the compiler will make an attempt to suppress all secondary consequences of it while continuing to process the rest of the program. In this way one translation process will often be able to give a clear indication of many errors in a program. (3) As far as the internal logic of the translator is concerned it is clear that a satisfactory solution along these lines will require the error detection to be completely integrated with the translation process.

* This paper was presented at the NordSAM conference in Stockholm, Aug. 1964.

The implementation of this philosophy in GIER ALGOL has been discussed at some length in section 4.6 of ref. 1. The result of this discussion is that we must recognize two major checking phases during the translation, an earlier one which deals with the delimiter structure of the source program, and a later one which checks the compatibility of the declarations given for the quantities and the way they are used to form expressions and statements.

The present paper deals entirely with the second of the two checking problems. It is in fact a much more complete description of the techniques mentioned briefly in sections 5.4 and 5.5 of ref. 1.

The problem.

The problem considered is the following: Given the text of the expressions and statements of the ALGOL program, and given also a complete description of the types and kinds of the identifiers and literal values, check that the usage of the identifiers and literal values conforms to their descriptions. For example, in the following statement:

$$A[i] := 5.7$$

the identifier A must denote an array of type **real** or **integer**, while i must be simple, or a procedure without parameters of type **real** or **integer**. In the following example:

$$p := B[k] := (\textbf{if } c \textbf{ then } d \textbf{ else } e) \div f$$

p and B must be both integer or both real; p must be simple, or a procedure identifier of a procedure whose body surrounds the statement; B must denote an array; c must be Boolean, k integer or real, d, e, and f, all integers, while each of k, c, d, e, and f, must be simple or a procedure without parameters.

A complete type checking might at first sight appear to be a rather complicated affair. However, as will be shown in detail below a neat and general solution may be obtained by performing a pseudo-evaluation of the expressions which proceeds like an actual evaluation as far as the combining of operators and operands is concerned, but which is only concerned with the types and kinds of operands, not with their values.

Pseudo-evaluation.

The pseudo-evaluation can only be performed at a stage when the expressions have been converted to a form which lends itself directly to an algorithmic evaluation. However, there is no reason why this form

should be the final one produced by the compiler; there are indeed very good reasons to perform the type check at a stage when the expressions are in the Reverse Polish form, irrespective of whether the final code uses this form or not. The point is that the transformation of the expressions to a form which makes good use of machine registers may again very conveniently be carried out by means of pseudo-evaluation. This is described by Jørn Jensen (ref. 2).

As is well known, the evaluation of arithmetic expressions written in Reverse Polish form will employ a stack to hold the values of those operands which are still waiting to be operated upon. Let us describe this stack as follows:

array *OPERAND STACK*[1: *some upper limit*]

and let us use an integer, *LAST USED*, to denote the value of the index of the current top element of this stack. The rules for evaluating an expression written in Reverse Polish form may now be stated as follows: Proceed through the expression from left to right in a strictly sequential fashion. When encountering an operand, place this at the top of the *OPERAND STACK* by performing the following operations:

LAST USED := *LAST USED* + 1;
OPERAND STACK[*LAST USED*] := *value*(*operand*);

When encountering an operator, perform the corresponding operation on the values found at the top of the stack and place the result of the operation also at the top of the stack.

The operators differ with respect to their use of the top operands. Unary operators ("negative", "not" and others) do not change *LAST USED*. Example: "negative" performs the following action:

OPERAND STACK[*LAST USED*]
:= −*OPERAND STACK*[*LAST USED*]

Binary operators (+, −, /, ×, ∨, ∧, =, etc.) remove two items from the top of the stack, but again place their result at the top, giving a net reduction of the size of the stack of one item. Example: The operator / produces the following action:

LAST USED := *LAST USED* − 1;
OPERAND STACK[*LAST USED*]
:= *OPERAND STACK*[*LAST USED*]/
OPERAND STACK[*LAST USED* + 1];

When performing a pseudo-evaluation, for checking purposes, the items held in the *OPERAND STACK* will not be values, but descriptions of the kinds and types of the corresponding operands. Likewise, the effect of operators in the input string should be expressed in terms of kinds and types. A few of the rules governing these effects are expressed in the following table:

Operator	Required types	Result type
Arithmetic (+, −, /, etc.)	**real** or **integer**	**real** or **integer**
Relational	**real** or **integer**	**Boolean**
Logical	**Boolean**	**Boolean**

To illustrate this approach, let us consider the following example of an expression

$$a \equiv b - c = d\,,$$

In Reverse Polish form this becomes

$$abc - d = \equiv\,,$$

Assuming the declarations

Boolean a; **integer** b, d; **real** c;

we get the following development of the stack during the pseudo-evaluation:

Input	*OPERAND STACK*[1]	*OPERAND STACK*[2]	*OPERAND STACK*[3]
a	Boolean		
b	Boolean	integer	
c	Boolean	integer	real
−	Boolean	real	
d	Boolean	real	integer
=	Boolean	Boolean	
≡	Boolean		

It should be noted that at the end of the pseudo-evaluation the top element of the *OPERAND STACK* holds a description of the type of the complete expression.

Checking during conversion.

In GIER ALGOL the type checking is done simultaneously with the conversion to Reverse Polish form. Therefore, although the above description represents the most transparent way of looking at the method, the practical use of it is better explained in terms of the actions of the conversion algorithm.

For converting from ALGOL source language structure to Reverse Polish form we use the algorithm described by Dijkstra (ref. 3). This method is based on the use of an operator stack and priorities associated with the individual operators. The algorithm scans the input sequentially from left to right. Every operand encountered in the input is immediately copied to the output. Binary operators in the input give rise to two distinct actions: first an unstacking of operators already placed in the operator stack is performed. This unstacking is done for the operators sitting at the top of the stack, until one is found whose priority is less than that of the operator coming in the input. The unstacking consists in moving the operator out of the stack and into the output. Secondly, the operator coming in the input is placed at the top of the operator stack.

When performing a checking of types simultaneously with the conversion to Reverse Polish form we will have both an *OPERAND STACK* and an *OPERATOR STACK* in action. The activity of the *OPERAND STACK* will in principle be controlled by the output produced during the conversion, as shown above. However, the simultaneity of conversion and checking actually permits a slight modification of the checking action, based on the following observation: At the moment when an operator from the input is placed in the operator stack, i.e. when any necessary unstacking has been done, the top element of the *OPERAND STACK* contains the left operand of that operator. Now, for checking purposes it is usually not necessary to have both of the operands of a binary operator at hand at the same time. It is therefore possible to complete the check on the left operand and remove its description from the *OPERAND STACK* at the moment when the operator is placed in the *OPERATOR STACK*.

As an illustration of this modified process we shall again follow the pseudo-evaluation of the above expression, but this time include the contents of the *OPERATOR STACK*. Also, the action will this time be controlled by the symbols of the original source text. We first give input, the contents of the two stacks, and the output, at each stage of the process. This is followed by a verbal description of each action. We use the abbreviations bo, in, and re, for Boolean, integer, and real types respectively.

The sequence of actions is as follows:

1. Operand *a* in input: Copy to output; place type in *OPERAND STACK*.

Action	Input	*OPERATOR STACK* [1]	*OPERATOR STACK* [2]	*OPERAND STACK* [1]	Output
1	*a*			bo	*a*
2	≡	≡			
3	*b*	≡		in	*b*
4	−	≡	−		
5	*c*	≡	−	re	*c*
6	=	≡		re	−
		≡	=		
7	*d*	≡	=	in	*d*
8	,	≡		bo	=
				bo	≡

2. Operator ≡ in input:
   A. Test for unstacking of operators (empty).
   B. Check type of top operand and remove. Place in *OPERATOR STACK*.
3. Operand *b* in input: Copy to output; place type in *OPERAND STACK*.
4. Operator − in input:
   A. Test for unstacking of operators (empty).
   B. Check type of top operand and remove. Place in *OPERATOR STACK*.
5. Operand *c* in input: Copy to output; place type in *OPERAND STACK*.
6. Operator = in input:
   A. Operator − from *OPERATOR STACK*: Check type of top operand and remove; place result type in *OPERAND STACK*; place in output.
   B. Check type of top operand and remove. Place in *OPERATOR STACK*.
7. Operand *d* in input: Copy to output; place type in *OPERAND STACK*.
8. Low priority operator in input:
   A. Operator = from *OPERATOR STACK*: Check type of top operand and remove; place result type in *OPERAND STACK*; place in output. Operator ≡ from *OPERATOR STACK*: Check type of top operand and remove; place result type in *OPERAND STACK*; place in output.
   B. Act on low priority operator in input.

This method of processing suggests the following way of looking at the operators in the source string: Every operator works on the value of

an expression which is placed immediately adjecent to it, to the left or right, or, in the case of binary operators, on both of these expressions. For the purpose of evaluation it is necessary that the operand on which the operator works is complete at the moment of operation. In a left to right scan this presents no problem for operators which operate to the left. Operators working to the right, on the other hand, must be moved to the other side of their operand. This is achieved with the aid of the operator stack. In this view the stack is a transmission device which moves operators from where they are written in the input text to the right side of their operands. Binary operators have a double effect, one occurring at the moment they are encountered in the input string, the other at the moment they are squeezed out of the operator stack.

This way of looking at the problem is useful because it immediately extends to many delimiters of ALGOL 60 which are not usually thought of as operators, e.g. **then**.

Checking conditional expressions.

As a further illustration of the present approach, consider the problem of conditional expressions. In this construction we have to compare the types of two or more expressions, each of arbitrary complexity, written at places of the text which may be separated by other expressions. As an example consider the following:

if a **then** b **else if** $c = d$ **then** $e-f$ **else if** $g \wedge h$ **then** $j+k$;

Here we have to check that the expressions b, $e-f$, and $j+k$, are compatible and that a, $c=d$, and $g \wedge h$, are correct Boolean expressions. Using the above approach the solution to this problem is as follows. First, we observe that **then** may be regarded as an operator of low priority working to the left and removing its operand from the stack. Second, the delimiter **else** also works to the left, but has to check that the type of its left operand matches the types of any previous, parallel expressions. It therefore has to compare two elements at the top of the stack and remove one of them, like a binary operator. For this to work properly we must let the first **if** of the expressions, but not the one following **else**, place an initial dummy operand of type *UNKNOWN* at the top of the stack, and make sure that the whole expressions is terminated by an operator working like **else**. Assuming the following correct types:

Boolean a, g, h; **integer** b; **real** c, d, e, f, j, k;

and writing un for *UNKNOWN* type, the check action will develop as follows:

Action	Input	*OPERATOR STACK*	*OPERAND STACK*		Output
		[1]	[1]	[2]	
1	**if**		un		
2	*a*		un	bo	*a*
3	**then**		un		**then**
4	*b*		un	in	*b*
5	**else**		in		**else**
6	**if**		in		
7	*c*		in	re	*c*
8	=	=	in		
9	*d*	=	in	re	*d*
10	**then**		in	bo	=
11			in		**then**
12	*e*		in	re	*e*
13	−	−	in		
14	*f*	−	in	re	*f*
15	**else**		in	re	−
16			re		**else**
17	**if**		re		
18	*g*		re	bo	*g*
19	∧	∧	re		
20	*h*	∧	re	bo	*h*
21	**then**		re	bo	∧
22			re		**then**
23	*j*		re	re	*j*
24	+	+	re		
25	*k*	+	re	re	*k*
26	;		re	re	+
27			re		**endelse**

In actions 3, 11, and 22, the **then** in the input checks the type at the top of the *OPERAND STACK* and removes it, after squeezing higher-priority operators out of the stack in actions 10 and 21. In actions 5, 16, and 27, **else** or **endelse** check the compatibility of the two top elements of the stack, remove one element, and replace the top element with a description of the type of the complete expressions as known up to that point. Up to action 5 this type is *UNKNOWN*, from then on up to action 16 it is **integer**, while from that point on it is **real** if we adopt the rule that a conditional expression with mixed real und integer constituents has the type real. The remaining actions bring nothing new. As usual the type of the complete expression is left as one element at the top of the *OPERAND STACK* when the algorithm has completed the scan.

Handling of errors.

The detection of errors is only half of the story of checking. An equally important part is the technique used for continuing the checking process without at the same time producing a large number of redundant error messages. The clue to a solution of this problem lies in treating errors in essentially the same manner as the regular language features, not as exceptions.

In GIER ALGOL a large part of the problem has already been taken care of in the earlier pass 3 where a thorough check of the delimiter structure has been made and erroneous spots of the program have been removed. During the type check of pass 6 we can therefore safely rely on the consistency of the delimiter structure. The principal part of the problem can therefore be solved by the introduction of an internal operand class, *UNDECLARED*. This is used as follows: Whenever the checking action detects a conflict between the class of an operand and the operator working on it, the operand is replaced by one of this internal class, at the same time as an error message is given. Further, operands of class *UNDECLARED* are accepted as correct in all contexts, to prevent redundant error messages.

Operands of class *UNDECLARED* are created already in pass 5 whenever identifiers are found to be, in fact, undeclared or when they are overdeclared, i.e. declared more than once within the same block. During pass 6 new operands of class *UNDECLARED* are only entered at the top of the *OPERAND STACK*.

A slight complication in the error handling is introduced by the modification of the basic conversion algorithm described above. In fact, an error detected in the left operand of a binary operator will have to be communicated through the operator stack. For this purpose we have to introduce a set of *UNDECLARED OPERATORs*, one for each possible priority number. An *UNDECLARED OPERATOR* in the *OPERATOR STACK* indicates that a binary operator of the same priority in the input has found an error when checking its left operand. Clearly, the *UNDECLARED OPERATORs* must accept any operand on their right without protest.

Classification and representation of operands.

The classes of operands to be distinguished during type checking are exactly those which differ with respect to the manner in which they are checked by operator actions. In GIER ALGOL altogether 25 classes are found useful. For the purpose of the following discussion these will be

numbered from 1 to 25. The meaning of each class is given in the following table and explained in the subsequent notes.

	Integer	Real	Boolean	Other
Simple	1	2	3	
Label				4
String				5
Nonsense				6
No-type procedure identifier				7
Non-formal array identifier	8	9	10	
Switch identifier				11
Unknown type				12
Undeclared				13
Formal array identifier	14	15	16	
Identifier of procedure with parameters, for call and assignment	17	18	19	
Identifier of procedure without parameters, for call and assignment	20	21	22	
Identifier of procedure without parameters, for call only	23	24	25	

The classes 1, 2, 3, 4, 5, 7, and 11, correspond closely to ALGOL types and need no further explanation. Class 6 is an extra type introduced in GIER ALGOL to take care of a particular facility of the output procedures. The reason why classes 8, 9, and 10, are held separate from classes 14, 15, and 16, is that since the correspondence between formal and actual parameters is not established during translation, formal array identifiers can only be checked with respect to type while for non-formal array identifiers the number of subscripts can be checked as well. Operands of classes 8, 9, and 10, have a subscript number associated with them, for use in checking against the number of subscripts occurring in the subscripted variables where they are used. Class 12 is used while checking conditional expressions to indicate an operand whose type is so far unknown. Class 13, *UNDECLARED*, has been explained in the section on the handling of errors above. The need for the classes 17 to 25 arises out of the peculiar appearance of the identifier of type procedures in left parts of assignment statements. This causes the identifier of a type procedure to lead a double existence. Inside the procedure body it may denote a call of the procedure or indicate the assignment of the value, while within the block where the procedure is declared local, but outside the procedure body, it may only denote the call. The complete representation of identifiers of classes 7, 17, 18, and 19, also supplies the number of parameters of the procedure. This reflects the fact that in GIER AL-

GOL we only check the number of actual parameters of a procedure call or function designator, not the types of the individual parameters.

It should be added that the classification given above is the one actually used within GIER ALGOL and so reflects certain illogicalities of the detailed coding of that compiler. In particular, the distinction between formal and non-formal array identifiers has a logical parallel for procedure identifiers, but this does not appear in the classification. This particular instance is due to a special treatment of function designators and procedure calls having non-empty actual parameter lists which shall not be described further here. It in no way indicates a failure of the present approach to the checking problem.

For the effectiveness of the checking algorithms the representation of the operand classes within the machine is of decisive importance. The representations should be chosen in such a manner that the tests required by all the different operator actions can be made with a minimum of effort. For example, the test for the correctness of the operand of the operator **then** consists in deciding whether this operand belongs to one of the classes 3, 22, or 25. Similarly, to test for an operand of the operator / (division) we have to decide whether it belongs to one of the classes 1, 2, 20, 21, 23, or 24. The variety of the selections of classes required by the various operator actions suggests the following general solution of the representation problem: Each class is represented as a bit pattern. Each bit position is associated with one particular operator checking action and the bit in that position will be 1 in the patterns representing operands which are admissible in the corresponding context, otherwise 0. With this representation the check whether an operand is correct in a given context is reduced to examining one single bit in the pattern representing the operand. In GIER ALGOL the operand bit patterns are about 30 bits long.

Additional parameters belonging to certain operand classes, such as numbers of subscripts, must be held separately. In GIER ALGOL two full machine words are used to hold each item of the *OPERAND STACK*. One of these will hold the bit pattern, the other any additional parameter.

Example of error messages.

The program and the corresponding error report produced by the GIER ALGOL III compiler shown below will give an impression of the completeness of the error detection achieved by the above techniques. The program is specially written to display each particular error type distinguished in the messages. The error report may be compared with the

information given in appendix 4 of ref. 4. The information given in appendix 4 of ref. 5 may also be used if it is noted that error type indication numbers given in that text must be modified as follows to conform to the newer edition:

Error numbers 578 to 688: subtract 2 .
- - 691 - 736: - 1 .

Test 6.14c all error types

```
begin integer i; real r; Boolean b;
array a1[1:i+b], a2[2:4, 5:7];
switch s := if r×p0 < 0 then L else L2;
procedure p0; r := i÷r;
procedure p1(f); real f; b := i+r;
L: b := i > s;
b := b ∧ p0;
r := b+a1;
L2: r := b ∨ b;
i := p1(r);
setchar(s);
outcopy(r);
go to b;
i;
p1(a2/r);
r[i] := 5;
to drum(r);
i := if b then s else r;
p1(if b then b else i-i);
r := b := b ∨ b;
for i := b step 1 until r do
  a1[s] :=
  a2[2];
a1 := 0;
if r then go to L;
for p0 := 1, 2 do
  a2[b,i] := 0;
begin array q[b:1]; end;
p1(i,r);
p1;
end;
```

6. line 1 type 576
line 2 type 576
line 3 type 582
line 4 type 585
line 5 type 590
line 6 type 593
line 7 type 640
line 7 type 596
line 8 type 599
line 9 type 604
line 10 type 616
line 11 type 616
line 12 type 630
line 13 type 633
line 14 type 640
line 15 type 646
line 16 type 649
line 17 type 657
line 18 type 677
line 19 type 686
line 20 type 690
line 21 type 697
line 22 subscript 704
line 23 type 714
line 24 type 725
line 25 type 728
line 26 type 732
line 27 type 735
line 28 proc. call or ident. 790
line 29 proc. call or ident. 840

REFERENCES

1. P. Naur, *The Design of the GIER ALGOL Compiler*, BIT 3 (1963), 124–140 and 145–166.
2. J. Jensen, *Generation of Machinecode in ALGOL Compilers*, Proc. NordSAM 64, to appear in BIT.
3. E. W. Dijkstra, *ALGOL 60 Translation*, ALGOL Bulletin Supplement no. 10, Math. Centrum Amsterdam, Nov. 1961; Annual Review of Automatic Programming Vol. III, 327–356, Pergamon Press, London, 1963.
4. P. Naur (ed.), *A Manual of GIER ALGOL III*, Regnecentralen, Copenhagen, 1964.
5. P. Naur (ed.), *A Manual of GIER ALGOL*, Regnecentralen, Copenhagen, 1963.

REGNECENTRALEN, COPENHAGEN, DENMARK