Bruno Buchberger

AN EXTENTION OF ALGOL 60
1. Altering programs during execution time

To alter programs during execution time is very easy in programming languages of the assembler type. However, it is not possible in ALGOL-like programming languages. The absence of such a feature in these languages is a severe drawback for many practical applications, for instance the realisation of "learning programs" or the application of function descriptions resulting from symbol manipulation programs to concrete arguments. The removal of this defect is the concern of the present note, where we shall define a suitable extension of ALGOL 60, which in our opinion could serve as a model for analogous extensions of similar languages (like FORTRAN [6], PL/1 [7], ALGOL 68 [5]).

2. Informal description of the extension

The two main ideas of the present proposal for installing the desired feature in ALGOL 60 are:
1. We enable procedure identifiers to have a variable meaning which can be altered during execution of the program by a special assignment statement:

(1) \[ \text{proc} := \text{c}; \] where proc is a procedure identifier and c the identifier of an ALGOL data-entity (for instance an integer-array). The meaning of this statement should be the following: take the values(s) of c and consider them as a description of a program according to a certain code, transform this description into a machine language program part corresponding to a procedure-declaration and take this declaration as the declaration for proc in the further execution.

2. As an essential feature of a suitable code for describing programs we would propose that the values of c after some easy "editing" form an ALGOL procedure declaration for the desired program. The transformation to a machine language program then, essentially, can be realized by an application of the compiler already available for the concrete ALGOL-implementation. Thus, the central effect of the proposed new variant of the assignment-statement would be a call of the compiler during execution time of the program, a possibility which was realized also in Busse [1].

For the theoretical purposes of this note we shall use the following code for the description of ALGOL-programs: In assignment-statements of the form (1) use only the identifiers of one-dimensional integer-arrays on the right-hand side. Define once and for all an injective mapping
mapid: \( T \rightarrow N \)

(\( T \): set of ALGOL-numbers, identifiers, -logical values, -delimiters, and -operators,
\( N \): set of natural numbers).

Then, as "ALGOL procedure declaration described by \( c \)" take the one described by

\[
M = \text{mapid}^{-1}(c[1]) \ldots \text{mapid}^{-1}(c[i])
\]

if there exists a "suitable" (cf. (4.54b)) \( i \) with lower bound of \( c \leq 1 \leq i \leq \text{upper bound of } c \).

The only extension of the language now consists in the proposed interpretation of assignment-statements of the form (1), which in ordinary ALGOL 60 would lead to an error-message during execution time (see Lauer [2], p.4-25). On the other hand, statements of the form (1) are not excluded by the syntax of ordinary ALGOL 60, such that the proposed extension is syntactically invisible (see Lauer [2], (2.31) or Naur [4], 4.2.1.).

By this simple extension we are now in a position to compose every possible ALGOL-procedure (for instance in the form of a sequence of integer-numbers) during execution time of a program by suitably manipulating the values of the integer-array (in general the data-entity) \( c \). After the procedure is set up it can be transmitted to execution by simply giving the instructions

\[
f := c; \quad f(\langle\text{actual parameter list}\rangle);
\]

where \( f \) has to be some identifier whose declaration is a procedure declaration, or by

\[
f := c;
\]
and using the function designator \( f(\langle\text{actual parameter list}\rangle) \) in some expression.

For practical purposes, of course, the special code defined above would not be convenient. A practically interesting implementation would probably have to be based on well developed string manipulation features, with careful consideration of the amount of work given to the "editing" function (in our proposal the function map, cf. (4.54b)). Also, such a code would have to be standardized to guarantee compatibility of programs using this new possibility and written for different implementations.

By the proposed method the desired language feature is realized in a very general way, such that really every possible ALGOL-program can be composed and executed during execution of some control program. Compared with other methods (for instance the "compile-time facilities" in PL/1) the proposed extension has several advantages:

1. Firstly, for the interpretation of statements having the form (1) we have not to include a new, long program part into the compiler, but only to alter the translation of the ":=\) in the special case (1) by putting a call of the "editing" function and the compiler to the translated program.

2. After a program described by \( c \) is once compiled by execution of \( f:=c \), it can be called as often as desired by the identifier \( f \) in its compiled, quickly operating machine-language form.

3. After execution of a procedure thus compiled, control automa-
typically returns to the status where new procedures can possibly be composed.

3. Formal definition of the extension

We now formally describe the extension using the description method developed by the IBM-Laboratory, Vienna. For understanding the following at least a survey knowledge of the method as given in Lucas, Lauer, Stigleitner[3] and the formal definition of ALGOL 60 syntax and semantics given in Lauer [2] is necessary. We use many definitions and notational conventions of those reports without explicitly stating them.

We already remarked that a syntactical extension is not necessary.
As to the semantics, we change the ALGOL 60 interpretation given in Lauer [2] by changing (4.54) there to

$$(4.54) \text{int-assign-st}(t) =$$
$$\text{length}(s-lp(t)) = 1 \& \text{is-proc-den}(\text{den}_t^1) \& \text{is-id}(r_t)$$
$$\& \text{is-INT}(s-elem-s-da(\text{den}_{r_t}^1)) \rightarrow$$
$$\text{upd-dn}(n_{l_t}^1, \text{den});$$
$$\text{den} : \text{combine}(p_t, s-e, s-e(\text{den}_t^1));$$
$$p_t : \text{pre-pass-text}(\text{translate}(\text{map}(r_t)))$$

$T \rightarrow$ right-hand part of (4.54) in Lauer [2] unchanged,

where $l_t = \text{elem}(1) \cdot s-lp(t), r_t = s-rp(t), n = m(E), \text{den}_t = s_m(DN) .

(4.54a) \text{translate(text)} = \text{this should be a function which for every character string } \text{txt} = \text{char}_1 \ldots \text{char}_n \text{ (char}_i \in \mathbb{T} \text{ (i=1, \ldots ,n)},
\( \overline{T} \) is the set of numbers, logical values, identifiers, delimiters and operators in the fixed concrete representation of abstract ALGOL 60 programs, \( \text{txt} \) being a syntactically correct procedure declaration in the concrete representation) gives the corresponding abstract object \( \text{txt}' \) satisfying \( \text{i-proc-decl} \). Note that no procedure identifier for the procedure under study appears in \( \text{txt}' \). We can suppose that the function \( \text{translate} \) is already defined according to the practical situations where for the fixed concrete representation this function, essentially, is given by the compiler. An example of a formal definition of a similar function is given in Lucas et al. [3], p.3-26.

\[
\text{map}(id) = \begin{cases} 
\text{map}^{-1}(id_1) \ldots \text{map}^{-1}(id_k), \\
\text{if } i_1 \leq i_2 \text{ & } (\exists j)Q(id,j) \\
\text{undefined else,}
\end{cases}
\]

\[
id_k = \text{elem}(-i_1+k+1) \cdot s \cdot \text{value}(\text{den}_id),
\]

\[
i_1 = s \cdot \text{lbd} \cdot s \cdot \text{da}(\text{den}_id),
\]

\[
i_2 = s \cdot \text{ubd} \cdot s \cdot \text{da}(\text{den}_id),
\]

\[
Q(id,j) = (1 \leq i_2 \text{ & } \text{map}^{-1}(id_1) \ldots \text{map}^{-1}(id_j) \text{ is a procedure declaration of the concrete representation})
\]

\[
i = (\exists j)Q(id,j),
\]

(4.54c) \( \text{map}(\overline{T}) \) is an injective mapping yielding an integer number for every element \( \in \overline{T} \).

(4.54d) \( \text{combine}(o,s,p) = \text{PASS}/(o;s;p) \).

This concludes the formal definition of the extension.'

Let us call ALGOL 60 machine the machine whose language function (state transition function) \( \lambda \) is described by the definition in Lauer [2] and ALGOL 60' machine the machine whose language function
is described by the definition in Lauer [2] plus the supplement
given above.

We know, firstly, that the above extension does no harm, as we can
easily prove

Lemma 1: Every abstract program t yielding a sequence of states
\( \xi_1, \xi_2, \ldots \) such that for no state \( \xi_k \) (\( k \geq 1 \)) \( s \text{-in-t} (\xi_k) = \text{error} \)
for \( t \text{-in-t} (s \text{-c}(\xi_k)) \), if submitted to the interpretation by the
ALGOL 60 machine, also yields the same sequence if submitted to
the interpretation by the ALGOL 60' machine.

Let us define concrete(obj) to uniquely yield a characterising
txt for every abstract obj satisfying is-proc-decl(obj), such
that translate(concrete(obj))=obj (see Lauer [2], chapter 5). The
definitions of syntactical predicates in the concrete representa-
tion should be such that concrete(obj) satisfies the predicate
"procedure declaration" of the concrete representation whenever
is proc-decl(obj). Further for any abstract object P we define

\[ P' = s(P; \{s-n = k | is-OWN(s\text{-scope} \cdot k(P))\}) \]

i.e. \( P' \) is the same object as \( P \) with all unique names assigned to
OWN-variables deleted. So, in particular, if \( P \) satisfies
is-p-proc-decl, then \( P' \) satisfies is-proc-decl. As in the fol-
lowing we shall speak about several distinct states \( \xi, \xi', \xi_1, \xi_2, \ldots \)
we shall agree to denote the corresponding immediate components
by: \( DN = s-dn(\xi), UN = s-un(\xi), \ldots, DN' = s-dn(\xi'), UN' = s-un(\xi'), \ldots \)
\( DN_1 = s-dn(\xi_1), UN_1 = s-un(\xi_1), \ldots \). Further, \( den_c = c(E)(DN) \) and
\( den_x = f(E)(DN) \).
Our main task is to show

Lemma 2: Consider $P = (\langle s\text{-type}: \text{type} \rangle, \langle s\text{-par-list}: \text{par-list} \rangle, \langle s\text{-spec-pt}: \text{spec-pt} \rangle, \langle s\text{-body}: \text{statement} \rangle)$

with $\text{is-p-proc-decl}(P)$ and a state $\xi$ with $f(\xi) = n_f, c(\xi) = n_c$, $s\text{-da}(\text{den}_c) = (\langle s\text{-lbd}: I_1 \rangle, \langle s\text{-ubd}: I_2 \rangle, \langle s\text{-elem}: \text{INTG} \rangle)$ and $I_1 ≤ I ≤ I_2$,

$\text{mapid}^{-1}(\text{elem}(-I_1+2)\cdot s\text{-value}(\text{den}_c)) \ldots \text{mapid}^{-1}(\text{elem}(I_1+1)\cdot s\text{-value}(\text{den}_c)) = \text{concrete}(P')$ for a certain $I$.

Then the execution of $t = (\langle s\text{-lp}: I \rangle, \langle s\text{-rp}: c \rangle)$, satisfying the first condition of (4.54), yields a state $\xi'$ such that

(+) $s\text{-typ}\cdot n_f(DN') = \text{type}$, $s\text{-par-list}\cdot n_f(DN') = \text{par-list}$, $s\text{-spec-pt}\cdot n_f(DN') = \text{spec-pt}$, $s\text{-body}\cdot n_f(DN') = \text{statement'}$,

$UN' = UN + k$, $C' = d(C; t)$, where $tn(C) = \{t\}$ and $\tau(C) = \text{int-st}(t)$.

$k$ is the number of OWN-variables in statement. statement' differs from statement only in the unique names standing at the positions $s\cdot n\cdot k$ of statement, where $\text{is-OWN}(s\text{-scope}\cdot k(\text{statement}))$. These unique names differ from each other and from all unique names used for OWN-variables throughout the program and for other identifiers in the present environment. Further, $s(\xi') = s(\xi)$ for all composite selectors $s$ differing from the composite selectors mentioned in (+).

Proof: We first compute by straightforward application of the definitions given in Lucas et al. [3] and Lauer [2]

$\xi_1 = \Psi(\xi, \tau) = \nu(\nu(\xi; \tau = s\cdot c; \langle \tau\cdot s\cdot c; \nu(\text{int-assign-st}(t)); \langle s\cdot \text{rl}(c) \rangle \rangle))$.

$\xi_1$ is like $\xi$, with the exception that now $C_1 = \nu(C \cdot s(\langle s\text{-in}: \text{int-assign-st} \rangle, \langle s\text{-al}: I \rangle, \langle s\text{-rl}(c) \rangle))$.

Still $tn(C_1) = \{t\}$.

10
For the next step the new form of (4.54) is used:

\[ S_2 = \psi(S_1, t) = \phi_{\text{int-assign-stat}}(t, \delta(S_1, t, s-c), t, \Omega) = \]

\[ = \mu(\delta(S_1, t, s-c), \lambda(t, s-c; \psi(c_1; c_2)) \). \]

Thus, also \( S_2 \) differs from \( S \) only by the \( s-c \) component which now is \( \Omega \).

where \( c_1 = \langle s-in: \text{up-dn} \rangle, \langle s-al: \text{map} \rangle, \langle r_r: \langle s-in: \text{combine} \rangle, \langle s-al: \text{map} \rangle, \langle s-e: \text{den-} \rangle, \rangle, \langle s-r: \langle r_r, \text{elem(2)} \cdot s-al \rangle, \rangle, \langle r_r: \langle s-in: \text{prep-} \rangle, \langle s-al: \langle \text{translate (map(c)) } \rangle, \langle s-r: \langle r_r, \text{elem(1)} \cdot s-al \cdot r \rangle \rangle \rangle \rangle \rangle \).

Now, \( \text{translate (map(c))} = P^t \), as one can easily check. Note that \( \text{is-proc-decl(P)} \) and therefore \( \text{concrete(P)} \) satisfies the predicate "procedure-declaration" of the concrete representation.

\( \text{tn}(C_2) = \{ t_2 \} \), where \( t_2 = r_r \cdot t_r \). Further,

\[ S_3 = \psi(S_2, t_2) = \phi_{\text{prep-} \rangle, \langle r_r: \langle s-al: \langle \text{translate (map(c)) } \rangle, \langle s-r: \langle r_r, \text{elem(1)} \cdot s-al \cdot r \rangle \rangle \rangle \rangle \}

\[ = \mu(\delta(S_2, t_2, s-c), \lambda(t_2, s-c; \mu \{(\text{prep- } \langle P^t, \text{un} \rangle, \langle k(\text{un}) \cdot \text{un-name} \rangle \}, \langle \text{is-OWN(s-scope, s-ri: r-r, elem(1) \cdot s-al \cdot r \rangle) \} \). \]

Thus,

\[ C_3 = \langle S_2, S_3; \langle s-in: \text{prep- } \rangle, \langle s-al: \langle P^t \rangle, \langle s-r: \langle r_r, \text{elem(1)} \cdot s-al \cdot r \rangle \rangle, \langle r_1: \langle s-in: \text{un-name} \rangle, \langle s-r: \langle r_1, \text{elem(2)} \cdot s-al \rangle \rangle, \ldots \rangle, \langle r_k: \langle s-in: \text{un-name} \rangle, \langle s-r: \langle r_k, \text{elem(2)} \cdot s-al \rangle \rangle \rangle \rangle, \langle k \rangle \text{ such that } \text{is-OWN(s-scope, s-ri: r-r, elem(1) \cdot s-al \cdot r \rangle) \} \rangle \]

\( \text{tn}(C_3) = \{ t_1 \cdot r_2, \ldots, t_k \cdot r_2 \} \).

For further processing we take the instructions at the nodes \( t_1 \cdot r_2, \ldots, t_k \cdot r_2 \) in one special order omitting the straightforward
proof, that order does not influence the final result.

\[ \xi_4 = \Psi(3, r_1 \, \tau_2) = \Phi_{\text{un-name}}(d(3, r_1 \, \tau_2 \, s \cdot c), r_1 \, \tau_2, \langle r_1 \, \kappa_1 \circ \text{elem}(2) \circ s \cdot \text{al} \rangle) \]

\[ = \mu(d(3, r_1 \, \tau_2 \, s \cdot c), \langle \kappa_1 \circ \text{elem}(2) \circ s \cdot \text{al} \circ (r_1 \, \tau_2 \, r_1) \circ \text{n-c} \text{n}_{\text{UN}} \rangle); \]

\[ \langle s \cdot \text{un} \text{=} \text{UN} + 1 \rangle, \]

\[ C_4 = d(C_3, r_1 \, \tau_2, \kappa_1 \circ \text{elem}(2) \circ s \cdot \text{al} \circ \tau_2 \text{n}_{\text{UN}}), \text{UN} = \text{UN} + 1, \]

\[ \tau_n(C_4) = \{r_2, \tau_2, \ldots, r_k \circ \tau_2\}. \]

Proceeding in this way we finally obtain

\[ C_{3+k} = d(C_3, r_1 \, \tau_2; \ldots, r_k \circ \tau_2), \langle \kappa_1 \circ \text{elem}(2) \circ s \cdot \text{al} \circ \tau_2 \text{n}_{\text{UN}}, \ldots, \]

\[ \langle \kappa_1 \circ \text{elem}(2) \circ s \cdot \text{al} \circ \tau_2 \text{n}_{\text{UN} + k - 1} \rangle, \]

\[ \text{UN} = \text{UN} + k, \tau_n(C_{3+k}) = \{\tau_2\}. \]

In the next step the newly generated k unique names are attached to all OWN-variables occurring within the s-body component of P', thus yielding an object P'', which is like P except for the unique names attached to the k OWN-variables.

\[ \xi_{4+k} = \Psi(3, k \, \tau_2) = \Phi_{\text{un-name}}(d(3, k \, \tau_2 \, s \cdot c), \kappa_1 \circ \text{elem}(1) \circ s \cdot \text{al} \circ r \circ (r_2 \, r \circ r) \circ \text{s-c}: \]

\[ \mu(P'; \langle s \cdot \text{n} \circ \kappa_1 \text{n}_{\text{UN}}, \ldots, (s \cdot \text{n} \circ \kappa_1 \text{n}_{\text{UN} + k - 1}) \rangle) \]

We omit the easy calculations of the next two steps which yield

\[ \xi' = \xi_{6+k} = \mu(d(3, k \, \tau_2 \, s \cdot c), \langle s \cdot \text{dn} \circ \nu(DN; \langle n_x \mu(P''; \langle s \cdot e; s \cdot e(den_x) \rangle) \rangle) \rangle), \]

\[ C' = d(C \circ \tau), \text{UN} = \text{UN} + k, \text{DN} = \mu(DN; \langle n_x \mu(P''; \langle s \cdot e; s \cdot e(den_x) \rangle) \rangle). \]

Thus, s-type\_n_x(DN') = type, s-par-list\_n_x(DN') = par-list,

s-spec\_pt\_n_x(DN') = spec\_pt, s-body\_n_x(DN') = statement' ,

where statement' has the property described in Lemma 2, because the use of the instruction un-name steadily produces new unique names. This completes our proof.
Lemma 2, informally speaking, has the following significance: given any procedure-denotation \( \text{den}_\varphi \) for an identifier \( f \), \( \text{den}_\varphi \) consisting of a procedure-declaration and an environment component, we can generate this procedure-denotation by first declaring \( f \) as procedure identifier of any procedure (thus defining the environment) and then executing \( f := c \) at any place where \( f \) is declared, composing in \( c[1], \ldots, c[I] \) a description of the procedure declaration. The execution of \( f := c \) then generates a procedure-denotation for \( f \), which differs from \( \text{den}_\varphi \) only in the choice of unique names for the OWE-variables, which is realized so that no conflict with other variables may arise. It is also shown, that the execution of \( f := c \) has no other effects. How the description of the procedure-declaration in \( c[1], \ldots, c[I] \) has to be composed is given by the function concrete, whose effect has to be known to the programmer.

I want to thank G.A. Ososkov, V.P. Shirikov, and A.A. Khoshenko, with whom I had several discussions on the subject of this paper.

**Literature:**


Received by Publishing Department on March 19, 1971.