E5 - 5787

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 6o, 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 6o 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) proc := c; where proc is a procedure identifier and c the identifier of an ALGOL data-entity (for instance an <u>integer</u>-array). The meaning of this statement should be the following: take the values(s) of c and consider them as a desription of a program according to a certain code, transform this desription 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 <u>integer</u>-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 $c \leq 1 \leq i \leq$ 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 6o 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 6o, 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 <u>integer</u>-numbers) during execution time of a program by suitably manipulating the values of the <u>integer</u>-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; f(<actual parameter list>);

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

f:= c;

and using the function designator f(<actual parameter list>) 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 desribed 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-

tically returns to the status where new procedures can possibly be composed.

3. Formal definition of the extension

We now formally desribe the extension using the desription 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 6o 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 6o interpretation given in Lauer [2] by changing (4.54) there to

(4.54) <u>int-assign-st</u>(t) =
$\quad$ length(s-lp(t))=1 & is-proc-den(den_{l_t}) & is-id(r_t)
$\quad$ & is-INT(s-elem∘s-da(den_{r_t})) $\longrightarrow$

$\quad$ <u>upd-dn</u>(n_{l_t},den);
$\quad\quad$ den: <u>combine</u>(pt,s-e,s-e(den_{l_t}));
$\quad\quad\quad$ pt: <u>prepass-text</u>(translate(map(r_t)))

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

where l_t=elem(1)∘s-lp(t), r_t=s-rp(t), n_m=m(<u>E</u>), den_m=n_m(<u>DN</u>).

(4.54a) translate(text) = this should be a function which for every character string txt=$char_1 \ldots char_n$ ($char_i \in$ T (i=1,...,n),

$\overline{T}$... set of numbers, logical values, identifiers, delimiters and operators in the fixed concrete representation of abstract ALGOL 6o programs, txt being a syntactically correct procedure declaration in the concrete representation) gives the corresponding abstract object txt' satisfying is-proc-decl. Note that no procedure identifier for the procedure under study appears in txt'. We can suppose that the function 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.

(4.54b)

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

$id_k = elem(-i_1+k+1) \circ s\text{-}value(den_{id})$,

$i_1 = s\text{-}lbd \circ s\text{-}da(den_{id})$,

$i_2 = s\text{-}ubd \circ s\text{-}da(den_{id})$,

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

$i = (\iota j)Q(id,j)$,

(4.54c) mapid(τ) is an injective mapping yielding an <u>integer</u> number for every element $\tau \in \overline{T}$.

(4.54d) <u>combine</u>(o,s,p) = PASS: μ(o;⟨s:p⟩).

This concludes the formal definition of the extension.

Let us call ALGOL 6o machine the machine whose language function (state transition function) Λ is desribed by the definition in Lauer [2] and ALGOL 6o' 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 ξ_k $(k \geq 1)$ $s\text{-}in \cdot \tau(\xi_k) =$ <u>error</u> for $\tau \in tn(s\text{-}c(\xi_k))$, if submitted to the interpretation by the ALGOL 6o machine, also yields the same sequence if submitted to the interpretation by the ALGOL 6o' 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 representation 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

$$P' = \delta(P; \{s\text{-}n \circ \kappa \mid is\text{-}OWN(s\text{-}scope \cdot \kappa(P))\}),$$

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 following we shall speak about several distinct states $\xi, \xi', \xi_1, \xi_2, \ldots$ we shall agree to denote the corresponding immediate components by: $\underline{DN}$=s-dn (ξ), $\underline{UN}$=s-un(ξ),..., $\underline{DN}'$=s-dn(ξ'), $\underline{UN}'$=s-un(ξ'),... $\underline{DN}_1$=s-dn(ξ_1), $\underline{UN}_1$=s-un(ξ_1),... . Further, $den_c = c(\underline{E})(\underline{DN})$ and $den_f = f(\underline{E})(\underline{DN})$.

Our main task is to show

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

with is-p-proc-decl(P) and a state ξ with $f(\underline{E})=n_f, c(\underline{E})=n_c$, $s\text{-da}(den_c)=(\langle s\text{-lbd}:I_1\rangle,\langle s\text{-ubd}:I_2\rangle,\langle s\text{-elem}:INTG\rangle)$ and $I_1 \leq 1 \leq I \leq I_2$, $mapid^{-1}(elem(-I_1+2)\circ s\text{-value}(den_c))\ldots mapid^{-1}(elem(I_1+I+1)\circ s\text{-value}(den_c))$ = concrete(P') for a certain I.

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

(+) $s\text{-typ}\circ n_f(\underline{DN}')=type$, $s\text{-par-list}\circ n_f(\underline{DN}')=par\text{-list}$,
$s\text{-spec-pt}\circ n_f(\underline{DN}')=spec\text{-pt}$, $s\text{-body}\circ n_f(\underline{DN}')=statement'$,
$\underline{UN}'=\underline{UN}+k$, $\underline{C}'=\delta(\underline{C};\tau)$, where $tn(\underline{C})=\{\tau\}$ and
$\tau(\underline{C})=\underline{\text{int-st}}(t)$.

k is the number of OWN-varibales in statement. statement' differs from statement only in the unique names standing at the positions $s\text{-}n\circ\kappa$ of statement, where $is\text{-}OWN(s\text{-scope}\circ\kappa(statement))$. 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]

$\xi_1=\psi(\xi,\tau)=\mu(\delta(\xi;\tau\circ s\text{-}c);\langle\tau\circ s\text{-}c:\mu(\underline{\text{int-assign-st}}(t);\langle s\text{-ri}:\Omega\rangle)\rangle)$.

ξ_1 is like ξ, with the exception that now

$\underline{C}_1=\mu(\underline{C};\langle\tau:(\langle s\text{-in}:\underline{\text{int-assign-st}}\rangle,\langle s\text{-al}:\langle t\rangle\rangle,\langle s\text{-ri}:\Omega\rangle)\rangle)$.

Still $tn(\underline{C}_1)=\{\tau\}$.

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

$\zeta_2=\psi(\zeta_1,\tau)=\phi_{\underline{\text{int-assign-st}}}(t,\delta(\zeta_1;\tau\circ\text{s-c}),\tau,\Omega)=$

$=\mu(\delta(\zeta_1;\tau\circ\text{s-c});\langle\tau\circ\text{s-c}:\mu(ct;\langle\text{s-ri}:\Omega\rangle)\rangle).$

Thus, also ζ_2 differs from ζ only by the s-c component which now is $\underline{C}_2=\mu(\underline{C};\langle\tau:ct\rangle)$, where

$ct=(\langle\text{s-in}:\underline{\text{upd-dn}}\rangle,\langle\text{s-al}:\langle n_r\rangle\rangle,\langle r:(\langle\text{s-in}:\underline{\text{combine}}\rangle,\langle\text{s-al}:\langle\Omega,\text{s-e},$
$\text{s-e}(\text{den}_r)\rangle\rangle,\langle\text{s-ri}:(\langle r,\text{elem}(2)\circ\text{s-al}\rangle)\rangle,\langle r:(\langle\text{s-in}:\underline{\text{prepass-text}}\rangle,$
$\langle\text{s-al}:\langle\text{translate}(\text{map}(c))\rangle\rangle,\langle\text{s-ri}:(\langle r\circ r,\text{elem}(1)\circ\text{s-al}\circ r\rangle)\rangle\rangle)\rangle).$

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

$\text{tn}(\underline{C}_2)=\{\tau_2\}$, where $\tau_2=r\circ r\circ\tau$. Further,

$\zeta_3=\psi(\zeta_2,\tau_2)=\phi_{\underline{\text{prepass-text}}}(P',\delta(\zeta_2;\tau_2\circ\text{s-c}),\tau_2,\langle r\circ r,\text{elem}(1)\circ\text{s-al}\circ r\rangle)$

$=\mu(\delta(\zeta;\tau_2\circ\text{s-c});\langle\tau_2\circ\text{s-c}:\mu((\underline{\text{prep-text-1}}(P',un);$
$\{\kappa(un):\underline{\text{un-name}}\mid\text{is-OWN}(\text{s-scope}\circ\kappa(P'))\});$
$\langle\text{s-ri}:\langle r\circ r,\text{elem}(1)\circ\text{s-al}\circ r\rangle\rangle)\rangle).$

Thus,

$\underline{C}_3=\mu(\underline{C}_2;\langle\tau_2:(\langle\text{s-in}:\underline{\text{prep-text-1}}\rangle,\langle\text{s-al}:\langle P'\rangle\rangle,$
$\langle\text{s-ri}:(\langle r\circ r,\text{elem}(1)\circ\text{s-al}\circ r\rangle)\rangle,$
$\langle r_1:(\langle\text{s-in}:\underline{\text{un-name}}\rangle,\langle\text{s-ri}:(\langle r_1,\kappa_1\circ\text{elem}(2)\circ\text{s-al}\rangle)\rangle)\rangle,\ \ldots,$
$\langle r_k:(\langle\text{s-in}:\underline{\text{un-name}}\rangle,\langle\text{s-ri}:(\langle r_k,\kappa_k\circ\text{elem}(2)\circ\text{s-al}\rangle)\rangle)\rangle)\rangle),$

κ_j such that is-OWN(s-scope$\circ\kappa_j$(P')) for $1\leq j\leq k$.

$\text{tn}(\underline{C}_3)=\{r_1\circ\tau_2,\ldots,r_k\circ\tau_2\}.$

For further processing we take the instructions at the nodes $r_1\circ\tau_2,\ldots,r_k\circ\tau_2$ in one special order omitting the straightforward proof, that order does not influence the final result.

$\xi_4=\Psi(\xi_3,r_1\cdot\tau_2)=\Phi_{\underline{\text{un-name}}}(\delta(\xi_3,r_1\circ\tau_2\circ\text{s-c}),r_1\cdot\tau_2,\langle r_1,\kappa_1\circ\text{elem}(2)\circ\text{s-al}\rangle)$
$=\mu(\mu(\delta(\xi_3;r_1\circ\tau_2\circ\text{s-c});\langle\kappa_1\circ\text{elem}(2)\circ\text{s-al}\circ(r_1\circ\tau_2-r_1)\cdot\text{s-c}:n_{\underline{UN}}\rangle);$
$\langle\text{s-un}:\underline{UN}+1\rangle),$

$\underline{C}_4=\mu(\delta(\underline{C}_3;r_1\cdot\tau_2);\kappa_1\circ\text{elem}(2)\circ\text{s-al}\circ\tau_2:n_{\underline{UN}}),\ \underline{UN}_4=\underline{UN}+1,$
$\text{tn}(\underline{C}_4)=\{r_2\circ\tau_2,\dots,r_k\circ\tau_2\}$. Proceeding in this way we finally obtain
$\underline{C}_{3+k}=\mu(\delta(\underline{C}_3;r_1\cdot\tau_2,\dots,r_k\circ\tau_2);\langle\kappa_1\cdot\text{elem}(2)\cdot\text{s-al}\cdot\tau_2:n_{\underline{UN}}\rangle,\dots,$
$\langle\kappa_k\cdot\text{elem}(2)\cdot\text{s-al}\cdot\tau_2:n_{\underline{UN}+k-1}\rangle),$

$\underline{UN}_{3+k}=\underline{UN}+k,\ \ \text{tn}(\underline{C}_{3+k})=\{\tau_2\}.$

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.

$$\xi_{4+k}=\Psi(\xi_{3+k},\tau_2)=\mu(\delta(\xi_{3+k};\tau_2\circ\text{s-c});\langle\text{elem}(1)\circ\text{s-al}\cdot r\circ(\tau_2-r\cdot r)\cdot\text{s-c}:\underbrace{\mu(P';\langle\text{s-n}\circ\kappa_1:n_{\underline{UN}}\rangle,\dots,\langle\text{s-n}\circ\kappa_k:n_{\underline{UN}+k-1}\rangle)}_{P''}\rangle),$$

We omit the easy calculations of the next two steps which yield
$\xi'=\xi_{6+k}=\mu(\delta(\xi_{5+k},\tau\circ\text{s-c});\langle\text{s-dn}:\mu(\underline{DN};\langle n_f:\mu(P'';\langle\text{s-e}:\ \text{s-e}(\text{den}_f)\rangle)\rangle)\rangle),$
$\underline{C}'=\delta(\underline{C};\tau),\ \underline{UN}'=\underline{UN}_{3+k}=\underline{UN}+k,\ \underline{DN}'=\mu(\underline{DN};\langle n_f:\mu(P'';\langle\text{s-e}:\text{s-e}(\text{den}_f)\rangle)\rangle).$
Thus, $\text{s-type}\circ n_f(\underline{DN}')=\text{type}$, $\text{s-par-list}\circ n_f(\underline{DN}')=\text{par-list}$,
$\text{s-spec-pt}\circ n_f(\underline{DN}')=\text{spec-pt}$, $\text{s-body}\circ n_f(\underline{DN}')=\text{statement}'$,
where statement' has the property described in Lemma 2, because the use of the instruction <u>un-name</u> steadily produces new unique names. This completes our proof.

Lemma 2, informally speaking, has the following significance: given any procedure-denotation den_f for an identifier f, den_f 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],..., c[I] a description of the procedure declaration. The execution of f:=c then generates a procedure-denotation for f, which differs from den_f only in the choice of unique names for the OWN-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], ..., 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.

<u>Literature:</u>

[1] H.G.Busse, Eine mögliche Erweiterung der Programmiersprache ALGOL, Elektronische Rechenanlagen, 8, Heft 2, 1966.

[2] P.Lauer, Formal definition of ALGOL 6o, Technical Report TR 25.o88, IBM Laboratory Vienna, 1968.

[3] P. Lucas, P.Lauer, H.Stigleitner, Method and notation for the formal definition of programming languages, Technical Report TR 25.o87, IBM Laboratory Vienna, 1968.

[4] P.Naur, Revised report on the algorithmic language ALGOL 6o,

[5] A.van Wijngaarden, Report on the algorithmic language ALGOL 68, Mathematisch Centrum Amsterdam, AS MR 1o1, 1969.

[6] V.P.Shirikov, Yazik FORTRAN, JINR, LVTA 1969.

[7] IBM System/36o Operating System, PL/1 Language Specifications, 1966, Form Nr. 79879-1.

Received by Publishing Department
on March 19, 1971.