Acta Informatica 2, 110–142 (1973)

On Correct Procedure Parameter Transmission in Higher Programming Languages*

Hans Langmaack

Received September 18, 1972

Summary. The paper starts with the observation that in ALGOL 60 no specifications for formal procedure parameters are prescribed, whereas ALGOL 68 demands complete specifications. As a consequence, no ALGOL 68 program accepted by the compiler can have wrong parameter transmissions at run time whereas ALGOL 60 programs may have them. The property of ALGOL 60 programs to have only correct parameter transmissions obviously is undecidable if all data, conditional statements, etc. have to be taken into consideration (Theorem 1) and it is unfair to demand that the compiler should decide that property by a finite process. Therefore, we investigate this question of decidability under a much fairer condition, namely without taking into consideration any data or conditions and by giving all procedure calls occurring in the same block "equal rights" (Section IV, p. 123). Even this fairer problem turns out to be algorithmically unsolvable, in general (Theorem 5), but it is solvable as soon as the programs do not have global formal procedure parameters (Theorem 3). Analogous answers can be given to the problems of formal equivalence of programs and of formal reachability, formal recursivity, and strong formal recursivity of procedures (Theorems 8–11). Procedures which are not strongly formally recursive have great importance in compilation techniques as is shown in Section X.

I. Introduction

This paper deals with the question whether formal parameters of procedures in high level programming languages should be specified or not. The situation is well known: In ALGOL 60 no specification is prescribed, whereas ALGOL 68 demands specifications for all formal parameters, even specifications for the formal parameters of formal procedures etc. must be given by the programmer. PL/1 takes a position in between: Formal parameters of non-formal procedures must be specified, but formal parameters of formal procedures cannot be specified. This means practically that PL/1 in this respect is closer to ALGOL 60 than to ALGOL 68. For, when translating a call of a non-formal or formal ALGOL 68 procedure the compiler is informed exactly about the specifications for all formal parameters. Best possible code can be implemented because superfluous actual data types need not be taken into consideration. Since no wrong parameter transmission can happen at run time no run time parameter checks (with respect

* The main results of this paper have been announced in an invited lecture given at the first annual congress of the Gesellschaft für Informatik (GI) in October 1971 in Munich.

to modes) need be implemented. Now, when translating a call of a formal ALGOL 60 or PL/1 procedure the compiler does not know any specifications for the formal parameters, so that even actual data types must be taken into account which at run time never occur. Because correct parameter transmission is not completely checked at compile time, run time parameter checks must be provided for.

This short discussion shows that, concerning parameter transmission, ALGOL 68 has clear advantages over the other languages mentioned. On the other hand, concerning parameter transmission, the definition of ALGOL 60 and PL/1 can well be justified if there is an algorithm which for any program at compile time firstly decides whether at run time wrong parameter transmissions might occur and which secondly detects the specifications for all formal procedure parameters. In the following we shall investigate the question in what sense and under which circumstances such an algorithm exists.

II. Language Limitations

In this paper we will discuss four higher level programming languages:

1. ALGOL 60 without specifications for formal parameters, called ALGOL 60-P (pure).

2. ALGOL 60 with specifications prescribed for formal parameters of non-formal procedures and denoted in that way indicated in the ALGOL 60 Report, called ALGOL 60-PL/1, as this language is PL/1 oriented.

3. ALGOL 60-PL/1 with additional specifications prescribed for formal parameters of formal procedures, called ALGOL 60-SF (specify formals). Formal parameters of formal procedures of formal procedures cannot be specified.

4. ALGOL 60 with complete specifications for formal parameters as in ALGOL 68, called ALGOL 60–68.

It is useful for our purposes to have a common frame for all these languages. We choose ALGOL 60 and trim the languages in such a way that they appear as successive restrictions of ALGOL 60-P. Different languages differ for us only by the method of indicating specifications for formal parameters.

In ALGOL 68 the formal parameters of formal procedures of formal procedures etc. have to be specified. Here, in general, mode declarers indicate modes structured like trees, trees which might even be infinite [5, 8]. Clearly, these infinite trees must be described in a finite manner.

We handle the parameter mechanism for procedure calls in that way which is given by the ALGOL 60 Report. Throughout this paper we understand the notion formal parameter in the sense of ALGOL 60. We do so even for the language ALGOL 60–68. The name for this language is justified because the method of indicating modes is modelled from the ALGOL 68 Report.

As an example we present one the same program Π^1 written in four different languages.

ALGOL 60-P:

```
begin ref real A;                                (concerning ref real and outreal
      proc P(X, Q);                              see the following modifications e)
           begin X := X + 1;                     and i))
                 if X < 5 then Q(X, P)
           end;
      A := 1;
      P(A, P);
      outreal A
end
```

ALGOL 60-PL/1:

```
begin ref real A;
      proc P(X, Q); ref real X; proc Q;
           begin X := X + 1;
                 :
                 etc. as above
                 :
```

ALGOL 60-SF:

```
begin ref real A;
      proc P(X, Q); ref real X;
                    proc (ref real, proc) Q;
           begin X := X + 1;
                 :
                 etc. as above
                 :
```

ALGOL 60-68:

```
begin ref real A;                                (in strict ALGOL 68 we would
      mode p = proc (ref real, p);               write ref real A = loc real;)
      proc P(X, Q); ref real X; p Q;
           begin X := X + 1;
                 :
                 etc. as above
                 :
```

For the aims of this paper it is not necessary to give complete definitions of the languages. It suffices to be acquainted with ALGOL 60. In order to allow proofs which are not swallowed up by formalities we impose restrictions and modifications on ALGOL 60:

a) Only proper procedures, no function procedures are allowed. For simplicity we write **proc** for the declarator **procedure**.

b) Value listing of formal parameters (in the sense of ALGOL 60) is prohibited.

c) Only identifiers are allowed as actual parameters of procedure statements.

d) Beside **begin** and **end** we have an additional pair of *statement braces* { }. They act as block-**begin** and block-**end** and we require that all procedure bodies are included in these braces. In this context, the new statement braces are called *body braces*.

e) We restrict the three data types of the ALGOL 60 Report **real**, **integer**, and **Boolean** to two, namely **real** and **Boolean**. We write **bool** for the latter. The unsigned numbers are of type **real**, the logical values **true** and **false** are of type **bool**. As a consequence we have only real and Boolean variables, no integer variables. For more clarity we use **ref real** and **ref bool** as declarators for real and Boolean variables and not **real** and **bool**. The types of constants remain **real** and **bool**.

f) We exclude arrays, subscripted variables, switches, and switch designators. Only identifiers, no unsigned integers are allowed as labels in front of label colons and as designational expressions behind **goto**.

g) The operators in arithmetic or Boolean expressions are $+$, $-$, $\times$, $/$, $\div$, $<$, $\leqq$, $=$, $\geqq$, $>$, $\neq$, $\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$, **if then else**. We further allow **abs**, **sign**, **entier** as unary prefix operators with their conventional meaning. The power operation $\uparrow$ and standard functions as *sin*, *cos*, etc. are excluded in order to avoid irrational numbers as results of operations.

h) We do not allow multiple assignment statements.

i) The input/output statements allowed are **inreal** ϱ, **outreal** ϱ, **inbool** β, **outbool** β where ϱ and β stand for real resp. Boolean variables.

j) In ALGOL 60-P we have no specifiers and the specification parts of procedure declarations are empty. In ALGOL 60-PL/1 the only specifiers allowed are **ref real**, **ref bool**, **label**, and **proc**. For ALGOL 60-SF and ALGOL 60–68 the formal parameters are given later in Definition 3. Restricted to programs with parameterless procedures all four languages are the same.

III. Syntactical and Formal Programs

We assume we have unambiguous context free grammars $\mathfrak{G}_P$, $\mathfrak{G}_{PL/1}$, $\mathfrak{G}_{SF}$, $\mathfrak{G}_{68}$ for ALGOL 60-P, -PL/1, -SF, -68 which are mere modifications of the grammar presented in the ALGOL 60 Report. Ambiguities still existing in the Report may be assumed to be remedied.

Definition 1. A *syntactical* program Π is a string of basic symbols (terminal symbols) which can be reduced to the axiom ⟨program⟩ by the formal rules of the respective grammar.

The property to be a syntactical program is decidable. By the help of a reduction sequence R from Π to ⟨program⟩ we can define which substrings in Π or (more exactly) which *occurrences* of substrings in Π are called *blocks*, *procedure declarations* (simply *procedures*), and *procedure bodies*. E.g. a substring is called a procedure declaration if it is reduced to the non-terminal symbol ⟨procedure declaration⟩ within a reduction sequence R from Π to ⟨program⟩. Because of the unambiguity of the grammar the definition is independent of the reduction sequence R chosen.

We consider as blocks not only proper blocks but also the whole program Π, procedure bodies, and so called *extended procedure bodies*. Extended bodies are extended by the formal parameter and specification part, while the declarator **proc** and the procedure identifier are excluded. Example:

$$\underbrace{\textbf{proc}\ P\underbrace{(X, Y);\ \textbf{ref real}\ X;\ \textbf{ref real}\ Y;\ \underbrace{\{X := Y\}}_{\text{procedure body}}}_{\text{extended procedure body}}}_{\text{procedure declaration or procedure}}$$

In a similar way we can define which substrings in Π are *identifiers*, *arithmetic expressions*, *Boolean expressions*, *assignment statements*, *procedure statements* etc. All these sets of substrings are decidable.

A syntactical program Π can also be considered to be a string

$$\Pi = Z_1 Z_2 \dots Z_n$$

where the symbols Z_i are *delimiters*, *constants*, or *identifiers*. If Z_i is an identifier then we denote by (i, Z_i) the *occurrence* of the identifier Z_i in the program $\Pi = Z_1 Z_2 \dots Z_n$. Occurrences of identifiers are *defining* or *applied*. Identifier occurrences in specification parts of procedures are ignored because they are redundant, in principle. It is well known how to establish in an ALGOL 60 program Π a relation δ between an occurrence (i, Z_i) of the identifier Z_i and a defining occurrence (j, Z_j) with $Z_i = Z_j$.

Definition 2. A syntactical program Π is called *formal*, if the relation δ is a function, totally defined on the set of all occurrences of identifiers in Π.

The property to be a formal program is decidable and δ is a computable function. If (i, Z_i) is an occurrence of the identifier Z_i then $\delta(i, Z_i) = (j, Z_j)$ is called the *associated defining occurrence* of this identifier. (i, Z_i) is also called a *bound occurrence*, bound by (j, Z_j). If we restrict δ to a block β in Π then δ is still a function, but not necessarily totally defined. If (i, Z_i) is an occurrence in β and $\delta(i, Z_i)$ is undefined in β, i.e. $\delta(i, Z_i)$ occurs outside β, then (i, Z_i) is called a *free occurrence* of the identifier Z_i in β.

Identifiers in a formal program Π may be *renamed*. Then $\Pi = Z_1 Z_2 \dots Z_n$ becomes $\tilde{\Pi} = \tilde{Z}_1 \tilde{Z}_2 \dots \tilde{Z}_n$. A renaming is called *admissible* if $\tilde{\Pi}$ is a formal program and if for all occurrences (i, Z_i) of identifiers in Π $pr_1(\delta(i, Z_i)) = pr_1(\delta(i, \tilde{Z}_i))$ holds. pr_1 is the *first projection* with $pr_1(j, Z_j) := j$. Two formal programs are called *identical* if they differ only by an admissible renaming of identifiers. A formal program is called *distinguished* if different defining occurrences of identifiers $(i, Z_i) \neq (j, Z_j)$ are denoted by different identifiers $Z_i \neq Z_j$. It is clear that in every class of identical formal programs there exists at least one distinguished program. All these properties defined above are decidable.

IV. Compilable Programs

In this section we should like to define, when a formal program is called to be correct with respect to compilation or simply compilable. Informally, we mean by this that any applied occurrence (i, Z_i) of an identifier, bound by the defining occurrence $\delta(i, Z_i) = (j, Z_j)$, is applied appropriately according to the definition.

We assign *modes* $\partial\sigma$ to occurrences of certain substrings σ in a formal program Π, and we do so at first for the language ALGOL 60-P. The possible modes are **real**, **bool**, **ref real**, **ref bool**, **label**, **proc** 0, **formal**, **proc** (**formal**, ..., **formal**). To every constant occurring in Π we assign the mode **real** resp. **bool** in the natural way. To every defining occurrence of a non-formal non-procedure identifier we assign the mode **ref real**, **ref bool**, or **label**, of a formal parameter we assign **formal**, of a non-formal procedure identifier we assign **proc** 0 resp. **proc** $(\partial\xi_1, \ldots, \partial\xi_\nu)$, where $\xi_1, \ldots, \xi_\nu$, $\nu \geqq 1$, are the formal parameters of the procedure. The mode $\partial(i, Z_i)$ of any occurrence of an identifier is defined by the mode $\partial\delta(i, Z_i)$ of the associated defining occurrence of the identifier.

The mode of any occurrence of a right hand expression of an assignment statement or of a Boolean expression in an if clause is defined by induction (In the further text we shall often drop the phrase "occurrence of"). These expressions may be thought to be constructed inductively with constants and identifiers as atomic expressions, arithmetic, relational, logical operators and **if then else** as function symbols, and (and) as brackets. Let α, β, γ be expressions with certain modes $\partial\alpha$, $\partial\beta$, $\partial\gamma$ if modes are defined.

Let ω be a unary arithmetic operator.

$$\partial\omega\,\alpha := \begin{cases} \textbf{real} & \text{if } \partial\alpha = \textbf{real} \text{ or } = \textbf{ref real} \text{ or } = \textbf{formal} \\ \text{undefined, otherwise.} \end{cases}$$

Let $\circ$ be a binary arithmetic operator.

$$\partial\alpha\circ\gamma := \begin{cases} \textbf{real} & \text{if } \partial\alpha = \textbf{real} \text{ or } = \textbf{ref real} \text{ or } = \textbf{formal} \\ & \quad \text{and } \partial\gamma = \textbf{real} \text{ or } = \textbf{ref real} \text{ or } = \textbf{formal} \\ \text{undefined, otherwise.} \end{cases}$$

For relational and logical operators modes are defined analogously. Let η be **if** β **then** α **else** γ.

$$\partial\eta := \begin{cases} \textbf{real} & \text{if } \partial\beta = \textbf{bool} \text{ or } = \textbf{ref bool} \text{ or } = \textbf{formal} \\ & \text{and } \partial\alpha = \textbf{real} \text{ or } = \textbf{ref real} \text{ or } = \textbf{formal} \\ & \text{and } \partial\gamma = \textbf{real} \text{ or } = \textbf{ref real} \text{ or } = \textbf{formal} \\ & \text{and not } \partial\alpha = \partial\gamma = \textbf{formal} \\ \textbf{bool} & \text{analogously, replace } \textbf{real} \text{ by } \textbf{bool} \\ \textbf{formal} & \text{if } \partial\beta = \textbf{bool} \text{ or } = \textbf{ref bool} \text{ or } = \textbf{formal} \\ & \text{and } \partial\alpha = \partial\gamma = \textbf{formal} \\ \text{undefined, otherwise} \end{cases}$$

$$\partial(\alpha) := \begin{cases} \textbf{real} & \text{if } \partial\alpha = \textbf{real} \text{ or } = \textbf{ref real} \\ \textbf{bool} & \text{if } \partial\alpha = \textbf{bool} \text{ or } = \textbf{ref bool} \\ \textbf{formal} & \text{if } \partial\alpha = \textbf{formal} \\ \text{undefined, otherwise.} \end{cases}$$

∂ is a computable function with a decidable domain of definition. Now we define for ALGOL 60-P:

Definition 3. A formal ALGOL 60-P program Π is called to be *correct with respect to compilation* or simply *compilable* if the following five conditions hold:

1) For any assignment statement in Π

$$\alpha := \gamma$$

where α is an identifier and γ is a right hand expression the following equations hold:

$$\partial\alpha = \textbf{ref real} \text{ or } = \textbf{formal},$$

$$\partial\gamma = \textbf{real} \text{ or } = \textbf{ref real} \text{ or } = \textbf{formal}$$

or analogously with **bool** instead of **real**.

2) For any goto statement

$$\textbf{goto}\ \alpha$$

where α is an identifier one of the equations

$$\partial\alpha = \textbf{label} \text{ or } = \textbf{formal}$$

holds.

3) For any procedure statement

$$\psi \text{ resp. } \psi(\alpha_1, \ldots, \alpha_\nu), \qquad \nu \geqq 1,$$

where $\psi, \alpha_1, \ldots, \alpha_\nu$ are identifiers one of the equations

$$\partial\psi = \textbf{formal} \text{ or } = \textbf{proc}\ 0$$

$$\text{resp. } \partial\psi = \textbf{formal} \text{ or } = \textbf{proc}(\underbrace{\textbf{formal}, \ldots, \textbf{formal}}_{\nu \text{ times}})$$

with the same $\nu \geqq 1$ as above holds.

4) For any Boolean expression β in an if clause of a conditional statement one of the equations

$$\partial\beta = \textbf{bool} \text{ or } = \textbf{ref bool} \text{ or } = \textbf{formal}$$

holds.

5) For any input/output statement

$$\textbf{inreal}\ \varrho,\ \textbf{outreal}\ \varrho,\ \textbf{inbool}\ \beta,\ \textbf{outbool}\ \beta$$

ϱ and β are non-formal identifiers with

$$\partial\varrho = \textbf{ref real} \text{ and } \partial\beta = \textbf{ref bool}.$$

The property to be a compilable ALGOL 60-P program Π is decidable. Our example program Π^1 in ALGOL 60-P is compilable as may be checked easily. Conditions 1)–5) are a precise formulation of the phrase "appropriate application of identifier occurrences"

The definition of the mode function ∂ for ALGOL 60-PL/1 changes in one respect only: The possible modes for formal parameters are **ref real**, **ref bool**, **label**, **proc**. Remember that we admit only identifiers as actual parameters.

The further definition of ∂ is exactly the same as for ALGOL 60-P. Because of the missing mode **formal** the definition could be formulated even simpler here. In Definition 3 only condition 3) is "strengthened":

3) For any procedure statement

$$\psi \text{ resp. } \psi(\alpha_1, \ldots, \alpha_\nu), \qquad \nu \geqq 1,$$

one of the following equations holds:

$$\partial\psi = \textbf{proc} \text{ or} = \textbf{proc}\ 0$$

$$\text{resp. } \partial\psi = \textbf{proc} \text{ or} = \textbf{proc}\,(\partial\xi_1, \ldots, \partial\xi_\nu)$$

with the same $\nu \geqq 1$ as above where $\xi_1, \ldots, \xi_\nu$ are the formal parameters of ψ and where for $\iota = 1, \ldots, \nu$ the following implications (*) are true:

$$\partial\xi_\iota \neq \textbf{proc} \succ \partial\xi_\iota = \partial\alpha_\iota$$

$$\partial\xi_\iota = \textbf{proc} \succ \partial\alpha_\iota \text{ is a procedure mode}$$

$$\textbf{proc}\ 0 \text{ or } \textbf{proc}(\ldots) \text{ or } \textbf{proc}.$$

The property to be a compilable ALGOL 60-PL/1 program is decidable. Our example program Π^1 in ALGOL 60-PL/1 is compilable. The modes of the actual parameters A, P are **ref real**, **proc**(**ref real**, **proc**) and of the corresponding formal parameters X, Q are **ref real**, **proc** so that the implications (*) are true.

The definitions for ALGOL 60-SF deviate from those in ALGOL 60-PL/1 only in the following respects: The possible modes for formal parameters are **ref real**, **ref bool**, **label**, **proc** 0, **proc**$(\mu_1, \ldots, \mu_\nu)$, where $\mu_1, \ldots, \mu_\nu$, $\nu \geqq 1$, stand for **ref real**, **ref bool**, **label**, or **proc**. Condition 3) in Definition 3 is strengthened further:

3) For any procedure statement

$$\psi \text{ resp. } \psi(\alpha_1, \ldots, \alpha_\nu), \qquad \nu \geqq 1,$$

the following equation holds:

$$\partial\psi = \textbf{proc}\ 0 \text{ resp. } \partial\psi = \textbf{proc}\,(\mu_1, \ldots, \mu_\nu)$$

with the same $\nu \geqq 1$ where $\mu_1, \ldots, \mu_\nu$ are the modes of the formal parameters of the non-formal or formal procedure identifier ψ and where for $\iota = 1, \ldots, \nu$ the following implications (**) are true:

$$\mu_\iota \text{ is different from any procedure mode} \succ \mu_\iota = \partial\alpha_\iota,$$

$$\mu_\iota = \textbf{proc}\ 0 \succ \partial\alpha_\iota = \textbf{proc}\ 0,$$

$$\mu_\iota = \textbf{proc} \quad \succ \partial\alpha_\iota \text{ is a procedure mode},$$

$$\mu_\iota = \textbf{proc}\,(\bar{\mu}_1, \ldots, \bar{\mu}_{\bar{\nu}}) \succ \partial\alpha = \textbf{proc}\,(\bar{\bar{\mu}}_1, \ldots, \bar{\bar{\mu}}_{\bar{\nu}})$$

$$\text{with the same number } \bar{\nu} \text{ of parameters}$$

$$\text{and } \bar{\mu}_{\bar{\iota}} = \textbf{proc} \succ \bar{\bar{\mu}}_{\bar{\iota}} \text{ is a procedure mode}$$

$$\text{and } \bar{\mu}_{\bar{\iota}} \neq \textbf{proc} \succ \bar{\mu}_{\bar{\iota}} = \bar{\bar{\mu}}_{\bar{\iota}}.$$

For short we may say that the modes $\partial\alpha_\iota$ and μ_ι must not be *contradictory*.

The property to be a compilable ALGOL 60-SF program is decidable. Our example program Π^1 in ALGOL 60-SF is compilable. The modes of the actual parameters X, P, A, P are **ref real**, **proc**(**ref real**, **proc**(**ref real**, **proc**)), **ref real**, **proc**(**ref real**, **proc**(**ref real**, **proc**)), and of the corresponding formal parameters are **ref real**, **proc**, **ref real**, **proc** (**ref real**, **proc**) so that the implications $(**)$ hold.

In ALGOL 60–68 the possible modes for identifiers are certain named trees. For our purposes, a *tree* T may be conceived as a non-empty set of finite strings, called *nodes*, over the natural numbers $\mathbb{N}$ with the following properties:

1. T is closed under initial segment relation, i.e. if st is in T then s is in T also.

2. If $t\nu$ with $\nu > 1$ is in T then $t(\nu-1)$ is in T also.

3. Any node t in T has at most finitely many immediate successors $t\nu$ in T with $\nu \in \mathbb{N}$.

A node t is called *maximal* (a *leaf*) if there is no immediate successor $t\nu$ in T with $\nu \in \mathbb{N}$. Non-maximal nodes are called *inner* nodes. A *mode tree* is a tree, the leaves of which are named by **ref real**, **ref bool**, **label**, or **proc** 0 and the inner nodes of which are named by **proc**. It is clear that all finite mode trees can be indicated in a 1—1 manner by finite function terms, socalled *fixed declarers*, generated by the calculus:

1) **ref real**, **ref bool**, **label**, and **proc** 0 are atomic fixed declarers (argument symbols).

2) If $\sigma_1, \ldots, \sigma_\nu$, $\nu \geqq 1$, are fixed declarers then **proc**$(\sigma_1, \ldots, \sigma_\nu)$ is a fixed declarer, too.

The indicating function

$$\mathfrak{J}_{\text{fix}} \,|\, \{\text{fixed declarers}\} \xrightarrow[\text{onto}]{1-1} \{\text{finite mode trees}\}$$

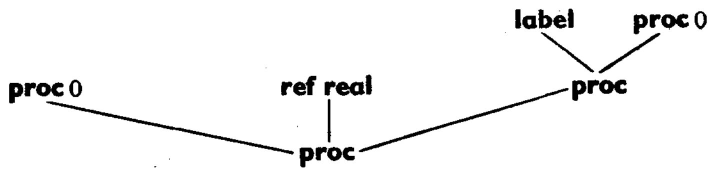
is defined inductively:

1) $\mathfrak{J}_{\text{fix}}$(**ref real**) := the single noded mode tree **ref real** etc. for the other atomic fixed declarers.

2) $\mathfrak{J}_{\text{fix}}$(**proc**$(\sigma_1, \ldots, \sigma_\nu)$) := the mode tree

$$\begin{tikzcd}[column sep=small] \mathfrak{J}_{\text{fix}}(\sigma_1) \arrow[dr, dash] & \ldots & \mathfrak{J}_{\text{fix}}(\sigma_\nu). \arrow[dl, dash] \\ & \textbf{proc} & \end{tikzcd}$$

E.g. the finite mode tree T



is indicated by the fixed declarer $\mathfrak{J}_{\text{fix}}^{-1}(T)$

proc (**proc** 0, **ref real**, **proc** (**label**, **proc** 0)).

Certain infinite mode trees can be indicated in a finite manner by the help of the mode declarations as it is done in the ALGOL 68 Report. We transfer this method from ALGOL 68 to ALGOL 60–68: In an ALGOL 60–68 program we allow to write down a finite system of $m \geqq 1$ "mode equations"

mode M$_1$ $= \tau_1$;

$\vdots$

mode M$_m$ $= \tau_m$;

The τ-s on the right hand side are (variable) *declarers* with the *mode indicants* $\mathbf{M}_1, \ldots, \mathbf{M}_m$ as additional atomic declarers (argument symbols). Example:

$$\begin{aligned}
&\textbf{mode } \mathbf{M}_1 = \textbf{proc}(\textbf{ref real}, \textbf{proc}(\mathbf{M}_3, \mathbf{M}_2), \textbf{label});\\
&\textbf{mode } \mathbf{M}_2 = \textbf{proc}(\textbf{proc}\,0, \mathbf{M}_3);\\
&\textbf{mode } \mathbf{M}_3 = \textbf{proc}(\textbf{proc}\,0, \textbf{proc}(\textbf{proc}\,0, \mathbf{M}_2));
\end{aligned}$$

We disallow a single mode indicant $\mathbf{M}_\mu$ as a right hand declarer.

Any chosen indicating function $\mathfrak{I}_{\text{var}}(\mathbf{M}_\mu) \in \{\text{mode trees}\}$ for mode indicants induces an extension

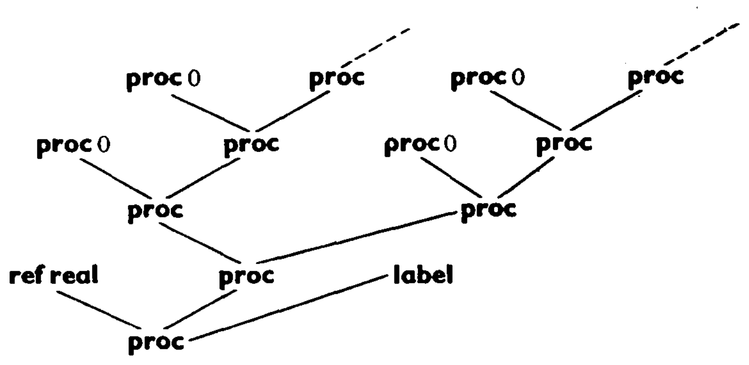
$$\mathfrak{I}_{\text{var}} | \{\text{declarers}\} \to \{\text{mode trees}\}$$

of $\mathfrak{I}_{\text{fix}}$ on the set of all declarers. It can be proven that every system of mode equations for the "variables" $\mathbf{M}_1, \ldots, \mathbf{M}_m$ has one unique system of mode trees $T_1, \ldots, T_m$ as its solution, such that the equations

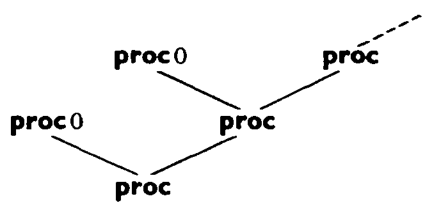
$$\begin{aligned}
\mathfrak{I}_{\text{var}}(\mathbf{M}_1) &= T_1 = \mathfrak{I}_{\text{var}}(\tau_1),\\
&\ \vdots\\
\mathfrak{I}_{\text{var}}(\mathbf{M}_m) &= T_m = \mathfrak{I}_{\text{var}}(\tau_m)
\end{aligned}$$

hold. The example equations above have the solution:

$T_1 =$



$T_2 = T_3 =$



The *mode* $\partial(i, Z_i)$ for a defining occurrence (i, Z_i) of a formal parameter in an ALGOL 60–68 program Π is that mode tree which is indicated by a (variable) declarer the mode indicants of which occur in a system of mode equations as described above. It is convenient to identify modes and their indicating declarers. Then, the further definition of the mode function ∂ is the same as for ALGOL 60-P. So we are able to introduce by finite means new and even infinite modes in an

ALGOL 60–68 program. Algorithms which effectively detect the identity of modes have been given in [5, 10, 12].

Condition 3) in Definition 3 is yet stronger compared to ALGOL 60-SF:

3) For any procedure statement

$$\psi \text{ resp. } \psi(\alpha_1, \ldots, \alpha_\nu), \qquad \nu \geqq 1,$$

the equation

$$\partial\psi = \mathbf{proc}\,0 \text{ resp. } \partial\psi = \mathbf{proc}(\partial\alpha_1, \ldots, \partial\alpha_\nu)$$

holds. Here we express that in ALGOL 60–68 coercions are not involved in parameter transmissions.

The property to be a compilable ALGOL 60-68 program is decidable because the identity of modes can effectively be detected. Our example program Π^1 in ALGOL 60–68 is compilable. The declarations of the actual parameters X, P, A, P are **ref real, p, ref real, p** and of the corresponding formal parameters are identically the same, namely **ref real, p, ref real, p**.

The following example Π^2 which we need later in Theorem 1 and 5 and Lemma 9 gives a further illustration of the notion of compilability:

```
begin proc $D(x, y)$; p$x$; q$y$; { };
      proc $M(x, y)$; p$x$; q$y$; {$x(y)$};
      proc $M1(x, y)$; p$x$; q$y$; {$x(D)$};
      proc $E(\eta)$; q$\eta$; {$\eta(E, D, D, M1)$};
      proc $\bar{E}(\xi, \alpha, \beta, \gamma)$; p$\xi$; r$\alpha, \beta, \gamma$; {$\gamma(\xi, \bar{E})$};
      $M(E, \bar{E})$ end
```

where **p** stands for **proc(proc)**, **q** for **proc(proc, proc, proc, proc)**, **r** for **proc(proc, proc)**. This program Π^2 is written in ALGOL 60-SF where formal parameters of formal procedures have to be specified. The program is compilable and, consequently, also compilable in ALGOL 60-P resp. ALGOL 60-PL/1 if we drop all specifications for formal parameters resp. parts of them. But the formal parameters cannot be specified in such a way that the program becomes compilable in ALGOL 60–68. Otherwise, the following equations for modes would hold:

(1)	$\partial E = \partial x^M$	because of $M(E, \bar{E})$
(2)	$\partial\eta = \partial y^M$	because of (1) and $x^M(y^M)$
(3)	$\partial\bar{E} = \partial y^M$	because of $M(E, \bar{E})$
(4)	$\partial\eta = \partial\bar{E}$	because of (2) and (3)
(5)	$\partial M1 = \partial\gamma$	because of $\eta(E, D, D, M1)$ and (4)
(6)	$\partial x^{M1} = \partial\xi$	because of $\gamma(\xi, \bar{E})$ and (5)
(7)	$\partial\xi = \partial E$	because of $\eta(E, D, D, M1)$ and (4)
(8)	$\partial x^{M1} = \partial E$	because of (6) and (7)
(9)	$\partial D = \partial\eta$	because of $x^{M1}(D)$ and (8)
(10)	$\mathbf{proc}(\partial x^D, \partial y^D) = \mathbf{proc}(\partial\xi, \partial\alpha, \partial\beta, \partial\gamma)$.	because of (9) and (4)

The last equation (10) is a contradiction. The superscripts M, $M1$, D in x^M, y^M, x^{M1}, y^{M1}, x^D, y^D have been written for better distinction.

We should not suppress the following remark concerning our definition of correctness with respect to compilation. The definition is based on a sort of local definition of appropriate application of identifier occurrences. If we would demand that the compiler should in addition trace all parameter transmissions, then the compiler could easily detect for this special program Π^2 that at run time the execution will lead to a wrong procedure call where actual and formal parameters do not harmonize:

$$\begin{array}{l} M(E,\bar{E}) \\ E(\bar{E}) \\ \bar{E}(E,D,D,M1) \\ M1(E,\bar{E}) \\ E(D). \end{array}$$

In ALGOL 60-SF the execution of Π^2 must stop here, because η has the mode $\mathbf{q}$ whereas D has the mode $\mathbf{r}$ with a different number of parameters. In ALGOL 60-PL/1 or -P the execution goes one step further:

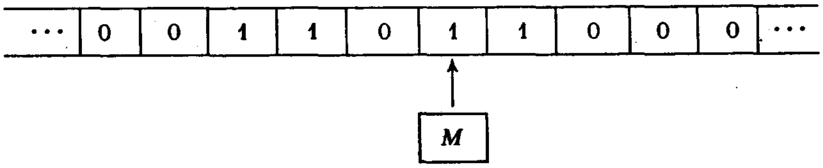
$$D(E,D,D,M1).$$

The procedure declaration D has two formal parameters whereas the procedure statement $D(E,D,D,M1)$ has four actual parameters so that the execution must stop.

On the other hand it is a crucial question whether we can fairly expect that a compiler performs tracing of all parameter transmissions. A compiler is an algorithm which among many other tasks has to give an answer "compilable" or "not compilable" for any submitted formal ALGOL program in a finite time. The tracing of parameter transmissions where all possible input data, all intermediate results, and all conditional statements must be taken into consideration is an algorithmically unsolvable problem: For we can easily prove

Theorem 1. There is no algorithm which for any given compilable ALGOL 60-P, -PL/1, or -SF program Π states whether there is a finite sequence d of input data (rational numbers or truth values) such that the execution of Π applied upon d will stop with a wrong procedure call.

In other words: It is undecidable, whether any compilable ALGOL 60-P, -PL/1, or -SF program Π has *actually occurring incorrect parameter transmissions.* The proof is standard and straight forward. Nevertheless, we present it here explicitely in order to show that we need different and more sophisticated techniques when we discuss the concept of formally correct parameter transmission later.



Proof. Let us consider a one tape Turing machine M over the alphabet 0, 1 with $m \geqq 1$ internal states $S_1, \ldots, S_{m_M}$. The associated program table has $2m_M$ rows and four columns.

$$\begin{array}{cccc}
S_1 & 0 & a_1 & S_{\mu_1} \\
S_1 & 1 & a_2 & S_{\mu_2} \\
\vdots & \vdots & \vdots & \vdots \\
S_{m_M} & 0 & a_{2m_M-1} & S_{\mu_{2m_M-1}} \\
S_{m_M} & 1 & a_{2m_M} & S_{\mu_{2m_M}}
\end{array}$$

The S_μ are internal states, the a_i are instructions *print* 0, *print* 1, *go left*, *go right*, or *stop*. It is well known that there is no algorithm which for any given Turing machine M and internal state S_{ν_M} proves or disproves the statement: If M is started in state S_1 and applied upon the empty tape (filled with zero digits only) then M will reach the state S_{ν_M}.

Now, we effectively construct for any given pair (M, S_{ν_M}) a compilable ALGOL 60-SF program Π^M.

```
begin ref real tape;
        proc $D(x, y)$; ...;
              ⋮
        proc $\tilde{E}(\xi, \alpha, \beta, \gamma)$; ...;
comment The procedure declarations are the five procedures $D, M, M1, E, \tilde{E}$
of program $\Pi^2$;
        tape := 0;
        goto $S_1$;
comment Now there follow conditional statements associated with the internal
states $S_\nu$, $\nu = 1, \ldots, \nu_M - 1, \nu_M + 1, \ldots, m_M$;
                   ⋮
        $S_\nu$: if tape − (entier tape) < 0.1 then
                  ⎧⟨dummy statement⟩;   if $a_{2\nu-1} = print\ 0$
                  ⎪tape := tape + 0.1;    if $a_{2\nu-1} = print\ 1$
            begin ⎨tape := tape/10;       if $a_{2\nu-1} = go\ left$
                  ⎪tape := tape × 10;     if $a_{2\nu-1} = go\ right$
                  ⎩goto STOP;              if $a_{2\nu-1} = stop$
                   goto $S_{\mu_{2\nu-1}}$ end
            else
                  ⎧tape := tape − 0.1;    if $a_{2\nu} = print\ 0$
                  ⎪⟨dummy statement⟩;   if $a_{2\nu} = print\ 1$
            begin ⎨tape := tape/10;       if $a_{2\nu} = go\ left$
                  ⎪tape := tape × 10;     if $a_{2\nu} = go\ right$
                  ⎩goto STOP;              if $a_{2\nu} = stop$
                   goto $S_{\mu_{2\nu}}$ end;
                   ⋮
comment The following statement is associated with the internal state $S_{\nu_M}$;
        $S_{\nu_M}$: $M(E, \tilde{E})$;
        STOP: outreal tape
end
```

Π^M works independently of any input data and, in essence, simulates the actions of the Turing machine M. The tape is represented by the real variable *tape*, a tape content by a decimal fraction, e.g. the tape content of the figure above by 110.11. The observed symbol is the digit immediately behind the decimal point. It is obvious that the execution of Π^M applied upon any input data sequence d will stop with a wrong procedure call if and only if the Turing machine M started in S_1 and applied upon the empty tape will reach the state S_{v_M}. So there cannot be any algorithm which fulfills the task described in Theorem 1. Q.e.d.

After this negative result in Theorem 1, we can hope at best to get a positive answer to the following problem: Does tracing of parameter transmissions become an algorithmically solvable task if we allow the compiler to disregard all data and conditional statements? This means more precisely: We do no longer consider procedures to be closed subroutines and do no longer assume that the execution of procedure statements is defined dynamically by dynamical applications of the copy rule. On the contrary, we consider procedures to be macros or open subroutines and we assume that a program Π with procedures is a textual abbreviation of a program Π^E without procedures. We shall call Π to have *formally correct parameter transmissions*, if Π^E is compilable (compare Definition 7). This concept of formally correct parameter transmission is independent of any execution of Π, neglects all data and conditions, and disregards the relative position of procedure statements within the main program or within the main part of a procedure body. The *main part* of a procedure body is that part which is outside of all procedures declared within the body. We say these procedure statements have "equal rights". In [4, 3, 6] there are algorithms which work under this assumption, but they are only sufficient, i.e. their answers are correct only if the answers read "the program has formally correct parameter transmissions".

Our proof of the undecidability of actually occurring incorrect parameter transmissions uses quite essentially data and conditional statements. The proof does not work if we would like to show the undecidability of formally incorrect parameter transmissions because every program Π^M has formally incorrect parameter transmissions. Our problem now is that Π^E is not finite in general.

V. Programs with Formally Correct Parameter Transmissions

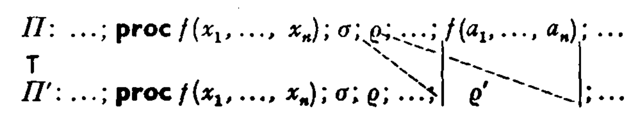
Definition 4. A formal program Π is called to be *partially compilable* if after replacement of all procedure bodies by empty ones the resulting program Π_e is compilable in the sense of Definition 3.

Definition 5. Let Π be partially compilable. A program Π' is called to *result from Π by application of the copy rule* ($\Pi \vdash \Pi'$) if the following holds:

Let $f(a_1, \ldots, a_n)$ be a procedure statement in the main program of Π. Let

$$\textbf{proc}\ f(x_1, \ldots, x_n);\ \underbrace{\sigma}_{\substack{\text{specification}\\ \text{part}}};\ \underbrace{\varrho}_{\substack{\text{procedure}\\ \text{body}}};$$

be the associated procedure declaration φ. Partial compilability of Π guarantees that the numbers n of actual and formal parameters are equal. We may assume that Π is distinguished. Then $f(a_1, \ldots, a_n)$ is replaced by a modified body ϱ', a so called *generated block*, where the formal parameters x_i occurring in ϱ are replaced by the corresponding actual parameters a_i. So we get Π'.



$$\begin{array}{l} \Pi\colon \ldots; \mathbf{proc}\ f(x_1, \ldots, x_n); \sigma; \varrho; \ldots; f(a_1, \ldots, a_n); \ldots \\ \top \\ \Pi'\colon \ldots; \mathbf{proc}\ f(x_1, \ldots, x_n); \sigma; \varrho; \ldots; \varrho'; \ldots \end{array}$$

Starting from Π' we can easily construct an identical and distinguished program Π'' if we rename all bound occurrences of identifiers in ϱ' which are bound within ϱ' (*local* to ϱ') by identifiers which do not yet occur in Π'.

The body braces $\{\}$ in $\varrho = \{\bar{\varrho}\}$ become so called *call braces* in $\varrho' = \{\bar{\varrho}'\}$. Let $\overset{+}{\vdash}$ and $\overset{*}{\vdash}$ be the transitive and transitive-reflexive closures of $\vdash$.

Lemma 1. If $\Pi \overset{+}{\vdash} \Pi'$ then Π' is a formal program. $\overset{+}{\vdash}$ resp. $\overset{*}{\vdash}$ are irreflexive resp. reflexive partial orderings in the set of formal programs.

Π' is not necessarily partially compilable even if Π is generally compilable. The programs Π^3

```
begin ref real a;
      proc p(x, y); {q(x)};
      p(a, p) end
```

resp.

```
begin ref real a;
      proc p(x, q); ref real x; proc q; {q(x)};
      p(a, p) end
```

fulfill the conditions of compilability in ALGOL 60-P resp. ALGOL 60-PL/1. But the following programs $\Pi^{3\prime}$ with $\Pi^3 \vdash \Pi^{3\prime}$ are not compilable, not even partially:

```
begin ref real a;
      proc p(x, q); {q(x)};
      {p(a)} end
```

resp.

```
begin ref real a;
      proc p(x, q); ref real x; proc q; {q(x)};
      {p(a)} end
```

Specifications in the manner of ALGOL 60-SF do not help either: The program Π^4

```
begin   ┌ ref real a;
      Δ │ proc p(q); proc(proc) q; {q(r)};
        │ proc f(x); proc(ref real) x; {x(a)};
        └ proc r(y); ref bool y; {y := true};
          p(f) end
```

is compilable, but program $\Pi^{4\prime}$

begin Δ; $\{f(r)\}$ **end**

with $\Pi^4 \vdash \Pi^{4\prime}$ is not partially compilable. On the other hand we can state the following:

Theorem 2. If Π is a compilable ALGOL 60–68 program and if $\Pi \overset{*}{\vdash} \Pi''$, then Π'' is compilable, too.

Proof. Let $\Pi \vdash \Pi'$. By Definition 5 all formal parameters x_i occurring in ϱ are replaced by the corresponding actual parameters a_i. As Π is compilable in ALGOL 60–68, x_i and a_i have identical mode trees. Therefore, as the application of x_i is appropriate in Π, the application of a_i in Π' must be appropriate, too. Q.e.d.

In the other languages this conclusion fails as in spite of compilability of Π, x_i and a_i do not necessarily have identical modes. In our example Π^4 q has the mode **proc**(**proc**), f has **proc**(**proc**(**ref real**)). These modes are not contradictory, and they are not identical either. In $\Pi^{4\prime}$ the modes of r **proc**(**ref bool**) and of x **proc**(**ref real**) are contradictory.

The proof of Theorem 2 for ALGOL 68 instead of ALGOL 60–68 would be more sophisticated because of the different parameter mechanisms and coercions prescribed by the ALGOL 68 and ALGOL 60 Reports.

Lemma 2. If $\Pi \vdash \Pi'$, then the procedure call $f(a_1, \ldots, a_n)$ in Π, mentioned in Definition 5, is uniquely determined by Π, Π'.

Lemma 3. If a diagram

$$\begin{tikzcd}
 & \Pi & \\
\Pi' \arrow[ur] & & \Pi'' \arrow[ul]
\end{tikzcd}$$

holds, then Π' and Π'' are identical or we can find a program Π''' with

$$\begin{tikzcd}
 & \Pi & \\
\Pi' \arrow[ur] & & \Pi'' \arrow[ul] \\
 & \Pi''' \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

if Π' and Π'' are partially compilable. An analogous lemma holds if we invert the arrows.

Two sequences

$$\Pi_0' \vdash \Pi_1' \vdash \cdots \vdash \Pi_{m'}' \quad \text{and}$$
$$\Pi_0'' \vdash \Pi_1'' \vdash \cdots \vdash \Pi_{m''}'', \quad m', m'' \geqq 0,$$

are called to *generate each other immediately* if $m'=m''$ and if the programs Π_i' and Π_i'' are identical for all indices i but one index i_0 with $0<i_0<m'$. The relation "*generate each other*" is defined to be the equivalent (transitive, reflexive, symmetric) closure of the relation just defined.

Lemma 4. If

$$\Pi=\Pi_0\vdash\Pi_1\vdash\cdots\vdash\Pi_m=\Pi' \quad\text{and}$$
$$\Pi=\tilde{\Pi}_0\vdash\tilde{\Pi}_1\vdash\cdots\vdash\tilde{\Pi}_{\tilde{m}}=\Pi'$$

then $m=\tilde{m}$ and both sequences generate each other.

Definition 6. Let Π be a formal program where { and } are used only as body braces. We call Π an *original* program. Then

$$E_\Pi:=\{\Pi'\mid\Pi\overset{*}{\vdash}\Pi'\}$$

is called the *execution* of Π. Programs Π' in E_Π, different from Π, are called *generated* programs.

For a generated program Π' in E_Π the following is true: In the main program of Π' outside of any innermost call brace pair all applications of identifiers are appropriate. Otherwisely Π' would not belong to E_Π. A program $\Pi'\in E_\Pi$ is maximal if and only if a) Π_e' (see Definition 4) is compilable and does not contain any procedure statement or b) there is exactly one innermost call brace pair in Π_e' with an inappropriate application of an identifier or c) Π_e' is equal Π_e with an inappropriate application of an identifier. Maximal programs Π' in E_Π of category b) or c) are exactly those, which are not partially compilable. Let $\Pi'\in E_\Pi$ be not partially compilable and different from Π, i.e. a maximal program of category b). Then there is exactly one preceeding program $p(\Pi')\in E_\Pi$ with $p(\Pi')\vdash\Pi'$.

E_Π contains a tree T_Π the nodes of which are exactly those programs in E_Π with at most one innermost call brace pair. We call T_Π the *execution tree* of Π. There is a bijection I_Π from E_Π onto the set $\mathfrak{T}_\Pi$ of all finite subtrees of T_Π, which contain as an element at most one program which is not partially compilable:

$$I_\Pi\mid E_\Pi\xrightarrow[\text{onto}]{1-1}\mathfrak{T}_\Pi.$$

The number of programs in $I_\Pi(\Pi')$ with $\Pi'\in E_\Pi$ is given by the number of call brace pairs in Π' plus one. If Π' is Π or partially compilable then

$$I_\Pi(\Pi')=\{\Pi''\mid\Pi''\in T_\Pi,\Pi\overset{*}{\vdash}\Pi''\overset{*}{\vdash}\Pi'\}.$$

If Π' is not Π and not partially compilable let

$$J_\Pi(\Pi')\text{ be }\{\bar{\Pi}\mid\bar{\Pi}\in T_\Pi,\Pi\overset{*}{\vdash}\bar{\Pi}\overset{*}{\vdash}p(\Pi')\}.$$

Then $I_\Pi(\Pi')$ is $J_\Pi(\Pi')$ added by one not partially compilable program $\Pi''\in T_\Pi$ with $\bar{\Pi}\vdash\Pi''$, $\bar{\Pi}$ maximal in $J_\Pi(\Pi')$ where Π'' results from $\bar{\Pi}$ by the "same" procedure statement which generates Π' from $p(\Pi')$. I_Π has the property

$$\Pi'\overset{*}{\vdash}\Pi''\ \gtrless\ I_\Pi(\Pi')\text{ is a subtree of }I_\Pi(\Pi''),$$

subtree in the following sense: $I_\Pi(\Pi')\subseteq I_\Pi(\Pi'')$ and if $I_\Pi(\Pi'')\ni\bar{\Pi}\overset{*}{\vdash}\bar{\bar{\Pi}}\in I_\Pi(\Pi')$ then $\bar{\Pi}\in I_\Pi(\Pi')$.

Definition 7. An original program Π is called to be *formally correct with respect to parameter transmissions* or to *have formally correct parameter transmissions* if all programs in E_Π (or T_Π) are partially compilable.

Lemma 5. If Π has formally correct parameter transmissions then at run time there is no actually occurring wrong procedure call.

Lemma 6. If Π has formally correct parameter transmissions then E_Π is a distributive lattice isomorphic to the lattice $\mathfrak{T}_\Pi$ of all finite subtrees of T_Π. I_Π is an isomorphism.

Concerning programs Π which have formally correct parameter transmissions we may give the following remark: Compilable programs $\tilde{\Pi}$ without procedures can be understood to be denotations for transformations $F_{\tilde{\Pi}}$ of finite data sequences d

$$F_{\tilde{\Pi}} | \{d\} \to \{d\}$$

where $\{d\}$ is the set of all finite data sequences. In general, $F_{\tilde{\Pi}}$ is only partially defined as $\tilde{\Pi}$ applied upon d might end with an error message or might run into an infinite loop. In these cases we say that $F_{\tilde{\Pi}}(d)$ is undefined. Let now Π' be a program in E_Π. We alter Π' into $\tilde{\Pi}'$ by eliminating all procedure declarations and by replacing all remaining procedure statements by

$$M\colon \textbf{goto } M.$$

$\tilde{\Pi}'$ is a program without procedures.

Now, if $\Pi' \vdash \Pi''$ then $F_{\tilde{\Pi}'} \subseteq F_{\tilde{\Pi}''} \subseteq \{d\} \times \{d\}$. Consequently, because E_Π is a lattice, the union

$$F_\Pi^* := \bigcup_{\Pi' \in E_\Pi} F_{\tilde{\Pi}'} \subseteq \{d\} \times \{d\}$$

is a well defined transformation $F_\Pi^* | \{d\} \to \{d\}$. So a program Π which has formally correct parameter transmissions can be understood to be a denotation for the transformation F_Π^* defined above (compare [1]).

VI. Programs without Global Formal Parameters

If $\Pi \vdash \Pi'$, then for all declarations Δ in Π we have identical copies Δ' in Π'. For all declarations Δ in the body ϱ we have additionally modified copies $\Delta'_{\varrho'}$ in ϱ' as parts of Π'. If $\Pi \overset{*}{\vdash} \Pi'$, i.e. $\Pi = \Pi_0 \vdash \Pi_1 \vdash \cdots \vdash \Pi_n = \Pi'$, $n \geqq 0$, then it is now clear, how to define when a declaration Δ' in Π' is called a *copy of a declaration* Δ in Π. Let Π be an original program and Π', Π'' programs in E_Π or T_Π. Declarations Δ' in Π' and Δ'' in Π'' are called *similar* if they are copies of the same declaration Δ in Π. A simple inductive argument shows

Lemma 7. Let d be a non-formal identifier occurring within the procedure body ϱ of a procedure declaration φ of an original program Π and let d have a declaration Δ. If φ' is a copy of φ in Π', then d has been replaced within φ' by the identifier d'' of a copy Δ'' of Δ. If the defining occurrence δd stands within ϱ then $\delta d''$ stands within the body ϱ' of φ', if δd stands outside ϱ then $\delta d''$ stands outside ϱ'.

Two nodes Π' and Π'' in T_Π are called *similar* if their innermost generated blocks (they are enclosed in call braces $\{\}$) ϱ' and ϱ'' differ by renaming of identifiers and if renamed indentifiers have similar declarations. The number of similarity classes for nodes in T_Π is limited by

$$M = P \cdot G^F + 1$$

where P is the number of non-formal procedure declarations, G is the number of defining occurrences of non-formal identifiers, and F is the number of defining occurrences of formal parameters in Π.

For a given original program Π we can effectively construct the smallest subtree U_Π of T_Π such that every maximal node in U_Π is maximal in T_Π or has a different similar predecessor in U_Π. Paths in U_Π have a length of at most $M+1$ nodes.

Let (i, x_i) be an applied occurrence of a formal parameter in program Π. If (i, x_i) occurs in the body ϱ of a procedure φ and if $\delta(i, x_i)$ occurs outside φ then x_i is called a *global formal parameter* of φ. Example:

$$\textbf{proc}\ p(x);\ \{\textbf{proc}\ q(y);\ \{\ldots x \ldots y \ldots\};\ \ldots\}$$

x is a global formal parameter of q.

Theorem 3. If an original program Π has no global formal procedure parameters then Π has formally correct parameter transmissions if and only if all programs in U_Π are partially compilable.

Corollary. For original programs Π without global formal procedure parameters it is decidable whether Π has formally correct parameter transmissions or not.

Proof of Theorem 3. Let Π' be a program in T_Π and not in U_Π. Then there is a maximal node Π' in U_Π with

$$\Pi' = \Pi_0 \vdash \Pi_1 \vdash \cdots \vdash \Pi_n = \Pi'', \qquad n > 0, \qquad \Pi_\nu \in T_\Pi.$$

We show that all Π_ν are partially compilable and that for every Π_ν there is a different similar node $\tilde{\Pi}_\nu$ in U_Π. This assertion is at least true for Π_0. Let it be true for $\Pi_{\nu-1}$, $0 \leqq \nu - 1 < n$. Then there is a different similar node $\tilde{\Pi}_{\nu-1}$ in U_Π. If $\tilde{\Pi}_{\nu-1}$ is maximal in U_Π it cannot be maximal in T_Π; otherwise, because of the partial compilability, the innermost call brace pair of $\tilde{\Pi}_{\nu-1}$ could not contain any procedure statement which would contradict $\Pi_{\nu-1} \vdash \Pi_\nu$. So in any case there is a different non-maximal node $\tilde{\tilde{\Pi}}_{\nu-1}$ in U_Π similar to $\Pi_{\nu-1}$. Both, $\Pi_{\nu-1}$ and $\tilde{\tilde{\Pi}}_{\nu-1}$, are partially compilable. Let

$$f(a_1, \ldots, a_n)$$

be the procedure statement within the main part of the innermost call brace pair of $\Pi_{\nu-1}$ which generates Π_ν. In $\tilde{\tilde{\Pi}}_{\nu-1}$ there is a corresponding procedure statement

$$\tilde{\tilde{f}}(\tilde{\tilde{a}}_1, \ldots, \tilde{\tilde{a}}_n)$$

where $f, \tilde{\tilde{f}}, a_1, \tilde{\tilde{a}}_1, \ldots, a_n, \tilde{\tilde{a}}_n$ have similar declarations. $\tilde{\tilde{f}}(\tilde{\tilde{a}}_1, \ldots, \tilde{\tilde{a}}_n)$ generates $\tilde{\tilde{\Pi}}'$ in U_Π:

$$\tilde{\tilde{\Pi}}_{\nu-1} \vdash \tilde{\tilde{\Pi}}'.$$

The declarations φ and $\tilde{\tilde{\varphi}}$ of f and $\tilde{\tilde{f}}$ are (eventually modified) copies of one the same declaration $\bar{\varphi}$ of $\bar{f}$ in Π. We have to check by which identifiers the global parameters d, occurring in $\bar{\varphi}$, have been replaced in φ and $\tilde{\tilde{\varphi}}$. d is non-formal by assumption. So, by Lemma 7 in both cases d has been replaced by identifiers having similar declarations. As a consequence the nodes Π_ν and $\tilde{\tilde{\Pi}}'$ are similar and Π_ν is partially compilable, too. We have therefore proven that Π has formally correct parameter transmissions. Q.e.d.

The following example Π^5 of an ALGOL 60-P program with global formal parameters shows that the assumption in Theorem 3 is essential:

```
begin   ⎧ proc l(μ); {μ(m, μ)};
        ⎪ proc m(φ); {φ := φ};
      Δ ⎨ proc f(x, y);
        ⎪     {proc q(ν); {x(ν)};
        ⎩      proc p(u, v); {q(v)}; y(m, p)};
        f(l, f) end
```

x is global formal parameter of procedure q. Π^5 has the following trees U_Π and T_Π

```
                 Π⁵ = begin Δ;
                            f(l, f) end
                            ⊤
            ...{proc q'(ν'); {l(ν')}};
                proc p'(u', v'); {q'(v')}; f(m, p')} ...
                            ⊤
            ...{proc q''(ν''); {m(ν'')}};
                proc p''(u'', v''); {q''(v'')}; p'(m, p'')} ...
                            ⊤
                       ...{q'(p'')} ...  ──────────┐
                            ⊤                      │
                       ...{l(p'')} ...,            │
                            ⊤              similar nodes
                       ...{p''(m, p'')} ...        │
                            ⊤                      │
   U_Π             Π₀ = ...{q''(p'')} ... ─────────┘
                            │
   . . . . . . . . . . . . .│ . . . . . .
   T_Π                      │
                   Π₁ = ...{m(q'')} ...
                            ⊤
                   Π₂ = ...{p'' := p''} ...
```

All programs in U_Π are partially compilable, nevertheless, Π has incorrect parameter transmission as in Π_2 p'' is not applied appropriately. The argumentation in the proof of Theorem 3 fails here because the global parameter x of q has been replaced by l in q' and m in q'', where l and m do not have similar declarations.

VII. Formally Equivalent Programs

In order to solve our decision problem for formally correct parameter transmissions we now might ask the following question: Is there an algorithm which transforms any program into an equivalent program without global formal parameters? Equivalence must be defined in such a way that it is invariant with respect to formally correct parameter transmissions.

Let Π be an original program. Let E_Π resp. T_Π be the execution resp. execution tree of Π. We form for any program $\Pi' \in E_\Pi$ the associated main program Π'_m by elimination of all procedure declarations and we replace every remaining procedure statement in Π'_m by a special symbol, say **call**, and term the result the *reduced main program* Π'_r of Π'.

Definition 7. $E_{r\Pi} := \{\Pi'_r \mid \Pi' \in E_\Pi\}$ is the *reduced execution* of Π. $T_{r\Pi} := \{\Pi'_r \mid \Pi' \in T_\Pi\}$ is the *reduced execution tree* of Π.

$T_{r\Pi}$ consists of exactly those reduced programs which contain at most one innermost call brace pair. The existence of not partially compilable programs in T_Π can be recognized in $T_{r\Pi}$ alone:

Π' in T_Π is not partially compilable $\Join$

1) Π'_r is maximal in $T_{r\Pi}$, and

2) the innermost call brace pair of Π'_r has an inappropriate application of an identifier or contains a **call**-symbol or $\Pi'_r = \Pi_r$ has an inappropriate application of an identifier or contains a **call**-symbol.

If we define now

Definition 8. Two original programs are called *formally equivalent* if their reduced execution trees are identical.

we can prove

Theorem 4. Let the original programs Π_1 and Π_2 be formally equivalent. Then Π_1 has formally correct parameter transmissions if and only if Π_2 has formally correct parameter transmissions, too.

In Definition 8 of formal equivalence of programs the term "reduced execution tree" could be replaced by "reduced execution" as the reader may prove. If two formally equivalent programs Π_1 and Π_2 have formally correct parameter transmissions, then they define the same transformation $F^*_{\Pi_1} = F^*_{\Pi_2}$.

VIII. Undecidabilities

We tried to construct algorithms which transform every ALGOL-program such that 1. global formal procedure parameters are eliminated and such that 2. the transformed program is formally equivalent to the original one. But all these constructions failed because for each of them we finally found example programs which did not fulfill the desired conditions. Therefore, we were led to the conjecture that our decision problem on formally correct procedure parameter transmission might be unsolvable, in general. We will now attack this conjecture by the help of Post's correspondence systems. Such a system has two alphabets

$$\mathfrak{A}=\{A, B\}, \quad A \neq B,$$
$$\bar{\mathfrak{A}}=\{\bar{A}, \bar{B}\}, \quad \bar{A} \neq \bar{B}, \quad \mathfrak{A} \cap \bar{\mathfrak{A}}=\emptyset.$$

So we have an isomorphism

$$\bar{\ } \mid \mathfrak{A} \xrightarrow[\text{onto}]{1-1} \bar{\mathfrak{A}}$$

which we continue to

$$\bar{\ } \mid \mathfrak{A}^* \xrightarrow[\text{onto}]{1-1} \bar{\mathfrak{A}}^* = \overline{\mathfrak{A}^*}.$$

We consider a production system

$$\Gamma=\{\gamma_1=(c_1, \tilde{c}_1)=(C_{11} \ldots C_{1n_1}, \tilde{C}_{11} \ldots \tilde{C}_{1\tilde{n}_1}),$$
$$\vdots$$
$$\gamma_m=(c_m, \tilde{c}_m)=(C_{m1} \ldots C_{mn_m}, \tilde{C}_{m1} \ldots \tilde{C}_{m\tilde{n}_m})\}$$

with

$$C_{ij} \in \mathfrak{A}, \qquad \tilde{C}_{ij} \in \bar{\mathfrak{A}}$$
$$\varepsilon \neq c_i \in \mathfrak{A}^*, \quad \varepsilon \neq \tilde{c}_i \in \bar{\mathfrak{A}}^*$$
$$n_i \geqq 1, \quad \tilde{n}_i \geqq 1, \quad m \geqq 1.$$

Definition 9. $\mathfrak{C}=(\mathfrak{A}, \bar{\mathfrak{A}}, \Gamma)$ is called a *correspondence system* of Post.

We consider non-empty sequences of indices $j_1, \ldots, j_r$ with $r \geqq 1$, $1 \leqq j_i \leqq m$.

Definition 10. A non-empty sequence of indices is a *solution* of $\mathfrak{C}$: $\Leftrightarrow$

$$\overline{c_{j_1} \ldots c_{j_r}} = \tilde{c}_{j_1} \ldots \tilde{c}_{j_r}.$$

Post's Theorem. The property "$\mathfrak{C}$ has a solution" is undecidable; in other words: Post's correspondence problem is unsolvable [8].

For any given correspondence system $\mathfrak{C}$ of Post we will now effectively construct a compilable ALGOL 60-P program $\Pi_{\mathfrak{C}}$ which fulfills the following

Lemma 8. $\mathfrak{C}$ has a solution $\Leftrightarrow$ $\Pi_{\mathfrak{C}}$ has not formally correct parameter transmissions.

As a consequence we have

Theorem 5. It is undecidable whether a compilable ALGOL 60-P program Π has formally correct parameter transmissions.

For given $\mathfrak{C}$ we construct program $\Pi_{\mathfrak{C}}$.

```
begin
comment The first part of Π_C is identical for all C;
proc D(x, y); { };
proc M(x, y); {x(y)};
proc M1(x, y); {x(D)};
proc E(η); {η(E, D, D, M1)};
proc Ē(ξ, α, β, γ); {γ(ξ, Ē)};
```

comment The second part of $\Pi_{\mathfrak{C}}$ is different for different $\mathfrak{C}$. For every j, $1 \leq j \leq m$ we have a procedure L_j. Within L_j we have procedures $C_{j1}[1], \ldots, C_{jn_j}[n_j]$ corresponding to the letters C_{ji} in c_j in the production $\gamma_j = (c_j, \tilde{c}_j)$. We have additional procedures $\tilde{C}_{j1}[1], \ldots, \tilde{C}_{j\tilde{n}_j}[\tilde{n}_j]$ corresponding to $\tilde{c}_j$. As the letters C_{ji} and $C_{ji'}$, $i \neq i'$ might be the same (A or B) we have to distinguish them by indices $[i] \neq [i']$;

```
    ⋮
proc $L_j(x, y)$;
    {proc $C_{j1}[1](\eta)$; $\{\eta(x, \varkappa_1\langle C_{j1}\rangle, \varkappa_2\langle C_{j1}\rangle, D)\}$;
     proc $C_{j2}[2](\eta)$; $\{\eta(C_{j1}[1], \varkappa_1\langle C_{j2}\rangle, \varkappa_2\langle C_{j2}\rangle, D)\}$;
       ⋮
     proc $C_{jn_j-1}[n_j-1](\eta)$; $\{\eta(C_{jn_j-2}[n_j-2], \varkappa_1\langle C_{jn_j-1}\rangle, \varkappa_2\langle C_{jn_j-1}\rangle, D)\}$;
     proc $C_{jn_j}[n_j](\eta)$; $\{\eta(C_{jn_j-1}[n_j-1], \varkappa_1\langle C_{jn_j}\rangle; \varkappa_2\langle C_{jn_j}\rangle, D)\}$;
     proc $\tilde{C}_{j1}[1](\xi, \alpha, \beta, \gamma)$; $\{\varkappa_3\langle \tilde{C}_{j1}\rangle(\xi, y)\}$;
     proc $\tilde{C}_{j2}[2](\xi, \alpha, \beta, \gamma)$; $\{\varkappa_3\langle \tilde{C}_{j2}\rangle(\xi, \tilde{C}_{j1}[1])\}$;
       ⋮
     proc $\tilde{C}_{j\tilde{n}_j-1}[\tilde{n}_j-1](\xi, \alpha, \beta, \gamma)$; $\{\varkappa_3\langle \tilde{C}_{j\tilde{n}_j-1}\rangle(\xi, \tilde{C}_{j\tilde{n}_j-2}[\tilde{n}_j-2])\}$;
     proc $\tilde{C}_{j\tilde{n}_j}[\tilde{n}_j](\xi, \alpha, \beta, \gamma)$; $\{\varkappa_3\langle \tilde{C}_{j\tilde{n}_j}\rangle(\xi, \tilde{C}_{j\tilde{n}_j-1}[\tilde{n}_j-1])\}$;
         $L_1(C_{jn_j}[n_j], \tilde{C}_{j\tilde{n}_j}[\tilde{n}_j])$; …; $L_m(C_{jn_j}[n_j], \tilde{C}_{jn_j}[\tilde{n}_j])$;
         $M(C_{jn_j}[n_j], \tilde{C}_{j\tilde{n}_j}[\tilde{n}_j])\}$;
     comment $\varkappa_1\langle C_{ji}\rangle$, $\varkappa_2\langle C_{ji}\rangle$ are denotations for $M$, $D$ if $C_{ji} = A$ and
     for $D$, $M$ if $C_{ji} = B$. $\varkappa_3\langle \tilde{C}_{ji}\rangle$ is a denotation for $\alpha$ if $\tilde{C}_{ji} = \bar{A}$ and for
     $\beta$ if $\tilde{C}_{ji} = \bar{B}$;
       ⋮
     $L_1(E, \bar{E})$; …; $L_m(E, \bar{E})$ end
```

Proof of Lemma 8. Any path in $T_{\Pi_{\mathfrak{C}}}$ starts with a non-empty sequence of calls of L_j:

$$\Pi_{\mathfrak{C}} \overset{L_{j_1}}{\vdash} \Pi_{\mathfrak{C}_{j_1}} \overset{L_{j_2}}{\vdash} \Pi_{\mathfrak{C}_{j_1 j_2}} \cdots \overset{L_{j_r}}{\vdash} \Pi_{\mathfrak{C}_{j_1 \ldots j_r}}.$$

We describe the structure of the node $\Pi_{\mathfrak{C}_{j_1 \ldots j_r}}$ with $r \geq 1$, $1 \leq j_\varrho \leq m$. Let us denote the corresponding strings

$$c_{j_1} \ldots c_{j_r} \in \mathfrak{A}^* \quad \text{and} \quad \tilde{c}_{j_1} \ldots \tilde{c}_{j_r} \in \bar{\mathfrak{A}}^*$$

with $\gamma_{j_\varrho} = (c_{j_\varrho}, \tilde{c}_{j_\varrho}) \in \Gamma$ by

$$D_1 \ldots D_N \quad \text{and} \quad \tilde{D}_1 \ldots \tilde{D}_{\tilde{N}}$$

with $D_i \in \mathfrak{A}$, $\tilde{D}_i \in \bar{\mathfrak{A}}$, $N = n_{j_1} + \cdots + n_{j_r}$ and $\tilde{N} = \tilde{n}_{j_1} + \cdots + \tilde{n}_{j_r}$ $(N \geq 1, \tilde{N} \geq 1)$. The necessary renamings of identifiers in $\Pi_{\mathfrak{C}_{j_1 \ldots j_r}}$ can be performed by raising the discriminating indices $[i]$ of

$$C_{ji}[i] \quad \text{and} \quad \tilde{C}_{ji}[i].$$

Compared with $\Pi_{\mathfrak{C}}$ $\Pi_{\mathfrak{C}_{j_1 \ldots j_r}}$ has the following additional procedure declarations (*)

```
proc $D_1[1](\eta)$; $\{\eta(E, \varkappa_1\langle D_1\rangle, \varkappa_2\langle D_1\rangle, D)\}$
proc $D_K[K](\eta)$; $\{\eta(D_{K-1}[K-1], \varkappa_1\langle D_K\rangle, \varkappa_2\langle D_K\rangle, D)\}$   for $K = 2, \ldots, N$
proc $\tilde{D}_1[1](\xi, \alpha, \beta, \gamma)$; $\{\varkappa_3\langle \tilde{D}_1\rangle(\xi, \bar{E})\}$
proc $\tilde{D}_K[K](\xi, \alpha, \beta, \gamma)$; $\{\varkappa_3\langle \tilde{D}_K\rangle(\xi, \tilde{D}_{K-1}[K-1])\}$   for $K = 2, \ldots, \tilde{N}$.
```

The main part of the innermost call brace pair of $H_{\mathfrak{C}_{i_1 \dots i_r}}$ has the following procedure statements (**)

$$\{\dots; L_1(D_N[N], \widetilde{D}_{\widetilde{N}}[\widetilde{N}]); \dots; L_m(D_N[N], \widetilde{D}_{\widetilde{N}}[\widetilde{N}]); M(D_N[N], \widetilde{D}_{\widetilde{N}}[\widetilde{N}])\}.$$

The proof for (*) and (**) can be given by a simple inductive argument. $H_{\mathfrak{C}_{i_1 \dots i_r}}$ is compilable. Therefore, the only chance to hit a maximal node on a path in $T_{H_{\mathfrak{C}}}$ is to call M for a first time. Repetitive calls of L_j lead only to an infinite path in $T_{H_{\mathfrak{C}}}$.

In a first case we assume that there is a greatest number h with

$$\bar{D}_N = \widetilde{D}_{\widetilde{N}}, \dots, \bar{D}_{N-h} = \widetilde{D}_{\widetilde{N}-h}$$
$$h \geq 0, \quad N-h \geq 2, \quad \widetilde{N}-h \geq 2.$$

Under this assumption we have a path

$$\begin{aligned} H_{\mathfrak{C}_{i_1 \dots i_r}} = \cdots & \{\dots; M(D_N[N], \widetilde{D}_{\widetilde{N}}[\widetilde{N}])\} \dots \\ & \qquad\qquad \top \\ \dots & \{D_N[N]\,(\widetilde{D}_{\widetilde{N}}[\widetilde{N}])\} \dots \\ & \qquad\qquad \top \\ \dots & \{\widetilde{D}_{\widetilde{N}}[\widetilde{N}]\,(D_{N-1}[N-1], \varkappa_1\langle D_N\rangle, \varkappa_2\langle D_N\rangle, D)\} \dots \\ & \qquad\qquad \top \\ \dots & \{M(D_{N-1}[N-1], \widetilde{D}_{\widetilde{N}-1}[\widetilde{N}-1])\} \dots \\ & \qquad\qquad \top \\ & \qquad\qquad \vdots \end{aligned}$$

which obviously can be prolonged to

$$\begin{aligned} & \qquad\qquad \vdots \\ & \qquad\qquad \top \\ \dots & \{M(D_{N-h-1}[N-h-1], \widetilde{D}_{\widetilde{N}-h-1}[\widetilde{N}-h-1])\} \dots \end{aligned}$$

We put $\check{N} = N-h-1$ and $\hat{N} = \widetilde{N}-h-1$ if h exists. In the second case, where h does not exist, we put $\check{N} = N$, $\hat{N} = \widetilde{N}$. Now, we have the following cases

$$\left.\begin{array}{ll} \text{a)} & \check{N} \geq 2,\ \hat{N} \geq 2 \\ \text{b)} & \check{N} = 1,\ \hat{N} \geq 2 \\ \text{c)} & \check{N} \geq 2,\ \hat{N} = 1 \\ \text{d)} & \check{N} = 1,\ \hat{N} = 1 \end{array}\right\} \text{ and } \bar{D}_{\check{N}} \neq \widetilde{D}_{\hat{N}}$$

$$\left.\begin{array}{ll} \text{e)} & \check{N} = 1,\ \hat{N} \geq 2 \\ \text{f)} & \check{N} \geq 2,\ \hat{N} = 1 \\ \text{g)} & \check{N} = 1,\ \hat{N} = 1 \end{array}\right\} \text{ and } \bar{D}_{\check{N}} = \widetilde{D}_{\hat{N}}$$

and in any case we can prolong

$$\begin{aligned} \dots & \{\dots M(D_{\check{N}}[\check{N}], \widetilde{D}_{\hat{N}}[\hat{N}])\} \dots \\ & \qquad\qquad \top \\ \dots & \{D_{\check{N}}[\check{N}]\,(\widetilde{D}_{\hat{N}}[\hat{N}])\} \dots \end{aligned}$$

Case g) means exactly that $j_1, \ldots, j_r$ is a solution of $\mathfrak{C}$. The other cases express the contrary. Case g) leads to a node in $T_{\Pi_{\mathfrak{C}}}$ which is not partially compilable:

$$\begin{array}{l}
\qquad\top \\
\ldots\{\widetilde{D}_1[1]\,(E, \varkappa_1\langle D_1\rangle, \varkappa_2\langle D_1\rangle, D)\}\ldots \\
\qquad\top \\
\ldots\{M(E, \bar{E})\}\ldots \\
\qquad\top \\
\ldots\{E(\bar{E})\}\ldots \\
\qquad\top \\
\ldots\{\bar{E}(E, D, D, M1)\}\ldots \\
\qquad\top \\
\ldots\{M1(E, \bar{E})\}\ldots \\
\qquad\top \\
\ldots\{E(D)\}\ldots \\
\qquad\top \\
\ldots\{D(E, D, D, M1)\}\ldots
\end{array}$$

D is not applied appropriately. All the other cases lead to a maximal node with a dummy statement between the innermost call brace pair:

case a)

$$\begin{array}{l}
\qquad\top \\
\ldots\{\widetilde{D}_{\overset{\frown}{N}}[\overset{\frown}{N}]\,(D_{\breve{N}-1}[\breve{N}-1], \varkappa_1\langle D_{\breve{N}}\rangle, \varkappa_2\langle D_{\breve{N}}\rangle, D)\}\ldots \\
\qquad\top \\
\ldots\{D(D_{\breve{N}-1}[\breve{N}-1], \widetilde{D}_{\overset{\frown}{N}-1}[\overset{\frown}{N}-1])\}\ldots \\
\qquad\top \\
\ldots\{\qquad\}\ldots
\end{array}$$

case b)

$$\begin{array}{l}
\qquad\top \\
\ldots\{\widetilde{D}_{\overset{\frown}{N}}[\overset{\frown}{N}]\,(E, \varkappa_1\langle D_1\rangle, \varkappa_2\langle D_1\rangle, D)\}\ldots \\
\qquad\top \\
\ldots\{D(E, \widetilde{D}_{\overset{\frown}{N}-1}[\overset{\frown}{N}-1])\}\ldots \\
\qquad\top \\
\ldots\{\qquad\}\ldots
\end{array}$$

case c)

$$\begin{array}{l}
\qquad\top \\
\ldots\{\widetilde{D}_1[1]\,(D_{\breve{N}-1}[\breve{N}-1], \varkappa_1\langle D_{\breve{N}}\rangle, \varkappa_2\langle D_{\breve{N}}\rangle, D)\}\ldots \\
\qquad\top \\
\ldots\{D(D_{\breve{N}-1}[\breve{N}-1], \bar{E})\}\ldots \\
\qquad\top \\
\ldots\{\qquad\}\ldots
\end{array}$$

case d)

$$\begin{array}{l}
\qquad\top \\
\ldots\{\widetilde{D}_1[1]\,(E, \varkappa_1\langle D_1\rangle, \varkappa_2\langle D_1\rangle, D)\}\ldots \\
\qquad\top \\
\ldots\{D(E, \bar{E})\}\ldots \\
\qquad\top \\
\ldots\{\qquad\}\ldots
\end{array}$$

case e)

$$\begin{array}{l}
\quad\;\top \\
\ldots\{\tilde{D}_{\hat{N}}[\hat{N}]\,(E,\varkappa_1\langle D_1\rangle,\varkappa_2\langle D_1\rangle,D)\}\ldots \\
\quad\;\top \\
\ldots\{M(E,\tilde{D}_{\hat{N}-1}[\hat{N}-1])\}\ldots \\
\quad\;\top \\
\ldots\{E\,(\tilde{D}_{\hat{N}-1}[\hat{N}-1])\}\ldots \\
\quad\;\top \\
\ldots\{\tilde{D}_{\hat{N}-1}[\hat{N}-1]\,(E,D,D,M1)\}\ldots \\
\quad\;\top \\
\ldots\left\{D\left(E,\begin{matrix}\bar{E}\text{ if }\hat{N}-1=1\\ \tilde{D}_{\hat{N}-2}[\hat{N}-2]\text{ if }\hat{N}-1\geqq 2\end{matrix}\right)\right\}\ldots \\
\quad\;\top \\
\ldots\{\qquad\qquad\}\ldots
\end{array}$$

case f)

$$\begin{array}{l}
\quad\;\top \\
\ldots\{\tilde{D}_1[1]\,(D_{\breve{N}-1}[\breve{N}-1],\varkappa_1\langle D_{\breve{N}}\rangle,\varkappa_2\langle D_{\breve{N}}\rangle,D)\}\ldots \\
\quad\;\top \\
\ldots\{M(D_{\breve{N}-1}[\breve{N}-1],\bar{E})\}\ldots \\
\quad\;\top \\
\ldots\{D_{\breve{N}-1}[\breve{N}-1]\,(\bar{E})\}\ldots \\
\quad\;\top \\
\ldots\left\{E\left(\begin{matrix}E\text{ if }\breve{N}-1=1\\ D_{\breve{N}-2}[\breve{N}-2]\text{ if }\breve{N}-1\geqq 2\end{matrix},\varkappa_1\langle D_{\breve{N}-1}\rangle,\varkappa_2\langle D_{\breve{N}-1}\rangle,D\right)\right\}\ldots \\
\quad\;\top \\
\ldots\left\{D\left(\text{or}\begin{matrix}E\\ D_{\breve{N}-2}[\breve{N}-2]\end{matrix},\bar{E}\right)\right\}\ldots \\
\quad\;\top \\
\ldots\{\qquad\qquad\}\ldots
\end{array}$$

We see therefore that $T_{\Pi_{\mathfrak{C}}}$ has a not partially compilable node if and only if there is a solution $j_1,\ldots,j_r$ of $\mathfrak{C}$. Q.e.d.

If we compare the proofs of Theorem 1 and Lemma 8 we see that the constructions and argumentations are quite different and that in Lemma 8 procedure statements and parameter transmissions play a much more important role whereas data and conditional statements do not and cannot play any role.

For better illustration we construct $\Pi_{\mathfrak{C}}$ for a concrete correspondence system $\mathfrak{C}$ with

$$\begin{aligned}\Gamma=\{\gamma_1&=(c_1,\tilde{c}_1)=(BA,\bar{B}),\\ \gamma_2&=(c_2,\tilde{c}_2)=(B,\bar{A}\,\bar{B})\}.\end{aligned}$$

$\mathfrak{C}$ has the solution 1, 2 as

$$\bar{c}_1\,\bar{c}_2=\overline{BA}\,\bar{B}=\bar{B}\bar{A}\,\bar{B}=\tilde{c}_1\,\tilde{c}_2$$

(case g)) whereas e.g. 1 is no solution as

$$\bar{c}_1=\overline{BA}\neq\bar{B}=\tilde{c}_1$$

(case c)). The second part of $\Pi_{\mathfrak{C}}$ looks as follows:

```
proc $L_1(x, y)$;
        {proc $B[1](\eta)$; $\{\eta(x, D, M, D)\}$;
         proc $A[2](\eta)$; $\{\eta(B[1], M, D, D)\}$;
         proc $\bar{B}[1](\xi, \alpha, \beta, \gamma)$; $\{\beta(\xi, y)\}$;
         $L_1(A[2], \bar{B}[1])$; $L_2(A[2], \bar{B}[1])$; $M(A[2], \bar{B}[1])\}$;
proc $L_2(x, y)$;
        {proc $B[1](\eta)$; $\{\eta(x, D, M, D)\}$;
         proc $\bar{A}[1](\xi, \alpha, \beta, \gamma)$; $\{\alpha(\xi, y)\}$;
         proc $\bar{B}[2](\xi, \alpha, \beta, \gamma)$; $\{\beta(\xi, \bar{A}[1])\}$;
         $L_1(B[1], \bar{B}[2])$; $L_2(B[1], \bar{B}[2])$; $M(B[1], \bar{B}[2])\}$;
$L_1(E, \bar{E})$; $L_2(E, \bar{E})$ end
```

A subtree of $T_{\Pi_{\mathfrak{C}}}$ is:

```
  $\Pi_{\mathfrak{C}}$ = ···$L_1(E, \bar{E})$...
                  ⊤
┌─$\Pi_{\mathfrak{C}_1}$ = ···{proc $B[1](\eta)$; $\{\eta(E, D, M, D)\}$;
│           proc $A[2](\eta)$; $\{\eta(B[1], M, D, D)\}$;
│           proc $\bar{B}[1](\xi, \alpha, \beta, \gamma)$; $\{\beta(\xi, \bar{E})\}$;
│           $L_1(A[2], \bar{B}[1])$; $L_2(A[2], \bar{B}[1])$; $M(A[2], \bar{B}[1])\}$...
│                              ⊤
= $\Pi_{\mathfrak{C}_{12}}$ = ···{proc $B[3](\eta)$; $\{\eta(A[2], D, M, D)\}$;
│           proc $\bar{A}[2](\xi, \alpha, \beta, \gamma)$; $\{\alpha(\xi, \bar{B}[1])\}$;
│           proc $\bar{B}[3](\xi, \alpha, \beta, \gamma)$; $\{\beta(\xi, \bar{A}[2])\}$;
│           $L_1(B[3], \bar{B}[3])$; $L_2(B[3], \bar{B}[3])$; $M(B[3], \bar{B}[3])\}$...
│                                                   ⊤
└─$\Pi_{\mathfrak{C}_1}$ = ···{...; $M(A[2], \bar{B}[1])\}$...        ...$\{B[3](\bar{B}[3])\}$...
                      ⊤                                  ⊤
           ...$\{A[2](\bar{B}[1])\}$...               ...$\{\bar{B}[3](A[2], D, M, D)\}$...
                      ⊤                                  ⊤
           ...$\{\bar{B}[1](B[1], M, D, D)\}$... ...$\{M(A[2], \bar{A}[2])\}$...
                      ⊤                                  ⊤
           ...$\{D(B[1], \bar{E})\}$...               ...$\{A[2](\bar{A}[2])\}$...
                      ⊤                                  ⊤
           ...{          }...                  ...$\{\bar{A}[2](B[1], M, D, D)\}$...
           partially compilable                          ⊤
                                               ...$\{M(B[1], \bar{B}[1])\}$...
                                                         ⊤
                                               ...$\{B[1](\bar{B}[1])\}$...
                                                         ⊤
                                               ...$\{\bar{B}[1](E, D, M, D)\}$...
                                                         ⊤
                                               ..$\{M(E, \bar{E})\}$...
                                                         ⊤
                                               ...$\{E(\bar{E})\}$...
                                                         ⊤
```

$$\begin{array}{l} \dots\{\bar{E}(E, D, D, MI)\}\dots \\ \quad\;\top \\ \dots\{MI(E, \bar{E})\}\dots \\ \quad\;\top \\ \dots\{E(D)\}\dots \\ \quad\;\top \\ \dots\{D(E, D, D, MI)\}\dots \\ \text{not partially compilable} \end{array}$$

The constructed programs $\Pi_{\mathfrak{C}}$ remain compilable in ALGOL 60-SF if we add appropriate specifications. All formal parameters x, ξ get the mode **proc(proc)**, y, η get **proc(proc, proc, proc, proc)**, α, β, γ get **proc(proc, proc)**. Lemma 8 remains true. The only difference in the proof is that inappropriate application of D reveals already in

$$\dots\{E(D)\}\dots$$

so that

$$\dots\{D(E, D, D, MI)\}\dots$$

cannot be a successor.

Theorem 6. It is undecidable whether a compilable ALGOL 60-SF (or ALGOL 60-PL/1) program Π has formally correct parameter transmissions.

On the other hand, it is impossible to add appropriate specifications such that all $\Pi_{\mathfrak{C}}$ become compilable ALGOL 60-68 programs and Lemma 8 remains true. If so then because of Theorem 2 there would not exist any solvable correspondence system. We can even prove directly for every $\Pi_{\mathfrak{C}}$:

Lemma 9. It is impossible to add appropriate specifications such that $\Pi_{\mathfrak{C}}$ becomes a compilable ALGOL 60-68 program.

Proof. If the contrary would hold we had the following equations for modes:

$$\left.\begin{array}{l} \partial E = \partial x^{L_1} = \partial C_{1n_1}[n_1] = \partial x^M \\ \partial \bar{E} = \partial y^{L_1} = \partial \tilde{C}_{1n_1}[\tilde{n}_1] = \partial y^M \end{array}\right\} \text{because of } L_1(E, \bar{E}),\ L_1(C_{1n_1}[n_1], \tilde{C}_{1\tilde{n}_1}[\tilde{n}_1]),$$
$$\text{and } M(C_{1n_1}[n_1], \tilde{C}_{1\tilde{n}_1}[\tilde{n}_1]).$$

As a consequence we can show up an equation

$$\mathbf{proc}(\partial\xi^{\bar{E}}, \partial\alpha^{E}, \partial\beta^{\bar{E}}, \partial\gamma^{\bar{E}}) = \mathbf{proc}(\partial x^{D}, \partial y^{D})$$

as we have done for the program Π^2 in Section IV. Contradiction! Q.e.d.

Concerning Theorems 5 and 6 the following remark is usefull: It is not satisfying that the execution of any program $\Pi_{\mathfrak{C}}$ applied upon any data sequence d runs into an infinite sequence of calls of procedure L_1 and never stops with an error message, say a wrong procedure call, although some programs $\Pi_{\mathfrak{C}}$ do not have formally correct parameter transmissions. The reader might suspect that the undecidable property of having formally correct parameter transmissions has little to do with real programming and real program execution. This is not true. Every program $\Pi_{\mathfrak{C}}$ can be augmented to a compilable program $\Pi_{\mathfrak{C}}^{a}$ by the help of appropriate input statements, conditional statements, labels, and goto statements such that $\Pi_{\mathfrak{C}}^{a}$ has the following property: $\Pi_{\mathfrak{C}}$ has not formally correct parameter transmissions if and only if $\Pi_{\mathfrak{C}}^{a}$ has not formally correct parameter transmissions and there is simultanously a data sequence d such that the execution of $\Pi_{\mathfrak{C}}^{a}$ applied upon d stops with a wrong procedure call (in other words: $\Pi_{\mathfrak{C}}^{a}$ has actually occurring incorrect parameter transmissions). So by Lemmas 5 and 8 and Post's Theorem we have another proof of Theorem 1.

IX. Application of the Proof Methods on other Problems

Theorem 7. There is no general algorithm which transforms any original program into a formally equivalent one without global formal parameters.

Proof. If there would be such an algorithm then because of Theorems 3 and 4 we would have a general decision process whether a compilable ALGOL 60-P program has formally correct parameter transmissions or not. This would contradict Theorem 5. Q.e.d.

Theorem 8. It is undecidable whether two original programs with formally correct parameter transmissions are formally equivalent.

Proof. For every $\mathfrak{C}$ we construct two different programs $\Pi_{\mathfrak{C}}^{1}$ and $\Pi_{\mathfrak{C}}^{2}$. For $\Pi_{\mathfrak{C}}^{1}$ the body $\{x(D)\}$ of $M1$ in $\Pi_{\mathfrak{C}}$ is replaced by $\{\ \}$, for $\Pi_{\mathfrak{C}}^{2}$ by $\{\textbf{ref real } A; A := A+1\}$. $\Pi_{\mathfrak{C}}^{1}$ and $\Pi_{\mathfrak{C}}^{2}$ are formally equivalent if and only if $\mathfrak{C}$ has no solution. So formal equivalence of ALGOL 60-P programs is undecidable. This is true even for ALGOL 60–68 programs (and consequently for ALGOL 60-SF and ALGOL 60-PL/1): For all formal parameters x, ξ we have to add the mode indicant **a**, for y, η we add **b**, for α, β, γ we add **c** with the following system of equations

$$\begin{aligned} &\textbf{mode a} = \textbf{proc}(\textbf{b});\\ &\textbf{mode b} = \textbf{proc}(\textbf{a}, \textbf{c}, \textbf{c}, \textbf{c});\\ &\textbf{mode c} = \textbf{proc}(\textbf{a}, \textbf{b}); \qquad \text{Q.e.d.} \end{aligned}$$

In the proof of Theorem 8 there necessarily occur infinite modes as the following equations must hold:

$$\begin{aligned} \partial C_{1n_1}[n_1] &= \textbf{proc}(\textbf{proc}(\partial C_{1,n_1-1}[n_1-1], \ldots, \ldots, \ldots))\\ &\ \vdots\\ \partial C_{11}\ [1] &= \textbf{proc}(\textbf{proc}(\partial x^{L_1}, \ldots, \ldots, \ldots)). \end{aligned}$$

So we have

$$\begin{aligned} \partial C_{1n_1}[n_1] &= \underbrace{\textbf{proc}(\ldots(\textbf{proc}}_{2n_1 \geqq 2 \text{ times}}(\partial x^{L_1}, \ldots, \ldots, \ldots)\ldots)\\ &= \textbf{proc}(\ldots(\textbf{proc}(\partial C_{1n_1}[n_1], \ldots, \ldots, \ldots)\ldots), \end{aligned}$$

an equation which cannot be fulfilled by finite modes alone. Therefore, we formulate the

Conjecture. Formal equivalence for ALGOL 60–68 programs becomes decidable if we restrict ourselves to programs with finite modes.

Definition 11. A procedure φ in an original program Π is called *formally reachable* if there is a node Π' in T_Π whose innermost generated block is the modified body of a copy of φ.

Theorem 9. It is undecidable whether a procedure φ in an original program with formally correct parameter transmissions is formally reachable.

Proof. $M1$ in $\Pi_{\mathfrak{C}}^{2}$ (see proof of Theorem 8) is formally reachable if and only if $\mathfrak{C}$ has a solution. This is true for all four languages. Q.e.d.

Definition 12. A procedure φ in an original program Π is called *formally recursive* if there are two different generated programs $\Pi' \overset{+}{\vdash} \Pi''$ in T_{Π} whose innermost generated blocks are modified bodies of copies of φ. φ is called *strongly formally recursive* if there are programs $\tilde{\Pi}' \vdash \Pi' \overset{*}{\vdash} \tilde{\Pi}'' \vdash \Pi''$ in T_{Π}, a copy $\tilde{\varphi}'$ of φ in $\tilde{\Pi}'$, and an identical copy $\tilde{\varphi}''$ of $\tilde{\varphi}'$ in $\tilde{\Pi}''$, such that the innermost generated block of Π' is a modified body of $\tilde{\varphi}'$ and the innermost generated block of Π'' is a modified body of $\tilde{\varphi}''$.

Theorem 10. It is undecidable whether a procedure in an original program with formally correct procedure parameter transmissions is formally recursive resp. strongly formally recursive.

Proof. In $\Pi_{\mathfrak{C}}^{2}$ we replace the body $\{\textbf{ref real}\ A;\ A := A+1\}$ by $\{M1(x, y)\}$ and we get $\Pi_{\mathfrak{C}}^{3}$. As $M1$ is a procedure, declared in the main program of $\Pi_{\mathfrak{C}}^{3}$, $M1$ is formally recursive if and only if $M1$ is strongly formally recursive. $M1$ is formally recursive if and only if $\mathfrak{C}$ has a solution. This is true for all four languages. Q.e.d.

Concerning Theorems 9 and 10 we have the analogous conjectures as the one formulated above.

Conjectures. Formal reachability, formal recursivity, and strongly formal recursivity of procedures in ALGOL 60–68 programs become decidable if we restrict ourselves to programs with finite modes.

By application of proof methods similar to those of Theorem 3 we may prove

Theorem 11. For programs without global formal procedure parameters it is decidable whether a procedure is formally reachable, formally recursive, or strongly formally recursive and whether two programs are formally equivalent.

X. Not Strongly Formally Recursive Procedures

The difference between formally recursive and strongly formally recursive procedures is important for compilation techniques, because those procedures which are not strongly formally recursive allow a simpler implementation than others. E.g. it is not necessary to reserve index- or displayregisters for them if the well known display method is used as an implementation method for procedures. Fixed storage places for simple and auxiliary variables local to a not strongly formally recursive procedure can be reserved among the fixed storage of the statically surrounding procedure so that we need an indexregister at most for this larger procedure.

If we conceive blocks as procedures without parameters called on the spot, then blocks are not strongly formally recursive. Not strongly formally recursive procedures can be handled like blocks.

Generated procedures, generated by the compiler as a substitute for complex expressions as actual parameters of procedure statements, may be formally recursive but are not strongly formally recursive. E.g.

$$\ldots;\ P(A + B[2], X);\ \ldots$$

has the compiled form

$$\ldots;\ \textbf{begin real proc}\ G;\ \{A + B[2]\};\ P(G, X)\ \textbf{end};\ \ldots$$

See [3], p. 119, where this special property of generated procedures is exploited in an ALGOL 60 compiler.

It is possible to handle for statement with two for list elements

$$\textbf{for}\ i := A\ \textbf{step}\ B\ \textbf{until}\ C,\ \bar{A}\ \textbf{step}\ \bar{B}\ \textbf{until}\ \bar{C}\ \textbf{do}\ S$$

as if two for statements

```
for $i := A$ step $B$ until $C$ do $S$;
for $i := \bar{A}$ step $\bar{B}$ until $\bar{C}$ do $S$
```

with identical controlled variables and controlled statements were given. Here the compiler is allowed to generate a procedure which is not strongly formally recursive:

```
begin
    proc $s$; $\{S\}$;
    for $i := A$ step $B$ until $C$ do $s$;
    for $i := \bar{A}$ step $\bar{B}$ until $\bar{C}$ do $s$
end
```

The blocks and the generated procedures are not strongly formally recursive because of the following

Theorem 12. A procedure φ without parameters whose identifier f occurs only in the main part of the procedure body (or in the main program) in which the procedure is declared is not strongly formally recursive.

Proof. Let φ have the form

$$\textbf{proc}\ f;\ \{\bar{\varrho}\}$$

declared in the main part of the body of procedure ψ in Π:

$$\Pi = \cdots \textbf{proc}\ g(x_1, \ldots, x_n);\ \underbrace{\{\ldots \textbf{proc}\ f;\ \{\bar{\varrho}\};\ \ldots\}}_{\bar{\varrho}^{\psi}} \cdots$$

Let us have a look at a path in the execution tree T_Π

$$\Pi = \Pi^0 \vdash \Pi^1 \vdash \Pi^2 \vdash \cdots$$

We may assume that all programs Π^i are distinguished and that successive programs

$$\Pi^i \vdash \Pi^{i+1}$$

are literally identical with the exception of that procedure statement in Π^i which is replaced by a modified procedure body in order to give Π^{i+1}. So, every declaration A in Π^i occurs also in Π^j for $j \geq i$ and there is a smallest number i_A such that the declaration A occurs in Π^{i_A}. We call Π^{i_A} the *associated program* of A and of the identifier of A.

Let the path have a node Π^{i_5} generated by a call of a copy φ'' of the procedure φ. Then, the path from Π^0 to Π^{i_5} has a structure

$$\begin{array}{l}
\Pi^0 = \cdots \textbf{proc}\, g(x_1, \ldots, x_n);\ \{\ldots \textbf{proc}\, f;\ \{\bar{\varrho}\};\ \ldots\} \ldots \\
\qquad\qquad\quad \mathsf{T}^* \\
\Pi^{i_1} = \cdots \textbf{proc}\, g'(x_1', \ldots, x_n');\ \underbrace{\{\ldots \textbf{proc}\, f';\ \{\bar{\varrho}'\};\ \ldots\}}_{\bar{\varrho}^{\psi}} \ldots \\
\qquad\qquad\quad \mathsf{T}^* \\
\Pi^{i_2} = \cdots \{\ldots;\ g'(a_1, \ldots, a_n);\ \ldots\} \ldots \\
\qquad\qquad\quad \mathsf{T} \\
\Pi^{i_3} = \cdots \underbrace{\{\ldots;\ \{\ldots \underbrace{\textbf{proc}\, f'';\ \{\bar{\varrho}''\}}_{\varphi''};\ \ldots\};\ \ldots\}}_{\bar{\varrho}_m^{\psi'}} \ldots \\
\qquad\qquad\quad \mathsf{T}^* \\
\Pi^{i_4} = \cdots\cdots \{\cdots;\ f'';\ \cdots\cdots\} \cdots\cdots \\
\qquad\qquad\quad \mathsf{T} \\
\Pi^{i_5} = \cdots\cdots \{\cdots\cdots;\ \{\bar{\varrho}_m''\};\ \cdots\cdots\} \cdots\cdots
\end{array}$$

Π^{i_3} is obviously the program associated to φ'' and to f''. As φ'' has no parameters the associated program of any identifier h occurring in $\bar{\varrho}_m''$ is Π^i with $i \leq i_3$ or $i \geq i_5$. Furthermore, h is different from f'' by assumption.

Now, let

$$h(a_1', \ldots, a_{n'}')$$

be a procedure statement in the main part of $\bar{\varrho}_m''$ and let Π^{i_5+1} result from Π^{i_5} by a call of $h(a_1', \ldots, a_{n'}')$:

$$\begin{array}{l}
\Pi^{i_5} = \cdots \{\ldots;\ h(a_1', \ldots, a_{n'}');\ \ldots\} \ldots \\
\qquad\qquad\qquad\qquad \mathsf{T} \\
\Pi^{i_5+1} = \cdots \{\ldots;\ \{\bar{\varrho}_m'''\};\ \ldots\} \ldots
\end{array}$$

Any identifier h' occurring in $\bar{\varrho}_m'''$ has an associated program Π^i with $i \leq i_3$ or $i \geq i_5$, as we can easily see. Furtheron, h' cannot be equal to f''. Otherwise, h would be the identifier of a procedure declared within $\bar{\varrho}_m^{\psi'}$ parallel to φ'' and f'' would occur in the body of h. This is impossible by assumption.

Iterating this argument we see that φ'' is never called a second time in the path. So φ is not strongly formally recursive. Q.e.d.

In the literature it is sometimes proposed to handle the blocks and generated procedures as if they were general procedures. This simplifies the whole translation and interpretation process, in principle. In favour of generating efficient object code procedures which are not strongly formally recursive should be processed differently. Unfortunately, Theorem 10 says that there does not exist any general algorithm which figures out exactly these special procedures. Therefore, theorems like Theorem 12 have a great importance for compilation techniques.

XI. Concluding Remarks

In a certain sense ALGOL 60 programs with procedures may be considered to be a sort of macro grammars which have been studied in the literature. In view of the results in [2], Theorem 9 looks surprising. In a further paper on elimination of global parameters and on normal forms for programs with procedures we shall investigate similarities and differences between programs and macro grammars [13].

The author should like to thank Prof. G. Hotz for many discussions, Dr. H. Feldmann for some hints, and the referees for their valuable critical remarks.

References

1. De Bakker, J. W., de Roever, W. P.: A Calculus for Recursive Program Schemes. MR 131/72, Mathematisch Centrum Amsterdam, February 1972.
2. Fischer, M. J.: Grammars with Macro-like Productions. Report No. NSF-22. Math. Ling. and Autom. Translation. Harvard Univ., Cambridge, Mass., May 1968.
3. Grau, A. A., Hill, U., Langmaack, H.: Translation of ALGOL 60. Handbook for Automatic Computation. Vol. I, Part b. Berlin-Heidelberg-New York: Springer 1967.
4. Hawkins, E. N., Huxtable, D. H. R.: A Multi-Pass Translation Scheme for ALGOL 60. Annual Review in Automatic Programming. Vol. III, 163–205. Oxford: Pergamon Press 1963.
5. Koster, C. H. A.: On Infinite Modes. ALGOL Bulletin No. 30, 86–89 (1969).
6. Ledgard, H. F.: A Model for Type Checking. Comm. ACM **15**, 956–966 (1972).
7. Naur, P. (Ed.) *et al.*: Revised Report on the Algorithmic Language ALGOL 60. Num. Math. **4**, 420–453 (1963).
8. Pair, C.: Concerning the Syntax of ALGOL 68. ALGOL Bulletin No. 31, 16–27 (1970).
9. Post, E. L.: A Variant of a Recursively Undecidable Problem. Bull. Am. Math. Soc. **52**, 264–268 (1946).
10. Scheidig, H.: Representation and Equality of Modes. Inf. Proc. Letters **1**, 61–65 (1971).
11. Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A.: Report on the Algorithmic Language ALGOL 68. Num. Math. **14**, 79–218 (1969).
12. Zosel, M.: A Formal Grammer for the Representation of Modes and its Application to ALGOL 68. Thesis, Univ. of Wash. 1971.
13. Langmaack, H.: On Procedures as Open Subroutines. Fachbereich Angew. Math. u. Informatik, Univ. des Saarlandes, Bericht A 73/04 (1973).

Prof. Dr. Hans Langmaack
Fachbereich Angewandte Mathematik
und Informatik
der Universität des Saarlandes
D-6600 Saarbrücken
Im Stadtwald
Bundesrepublik Deutschland