REVUE FRANÇAISE D'AUTOMATIQUE, INFORMATIQUE, RECHERCHE OPÉRATIONNELLE. INFORMATIQUE THÉORIQUE

PAUL FEAUTRIER

Post grammars as a programming language description tool

Revue française d'automatique, informatique, recherche opérationnelle. Informatique théorique, tome 9, n° R1 (1975), p. 43-72.

<http://www.numdam.org/item?id=ITA_1975__9_1_43_0>

R.A.I.R.O.
(9e année, avril 1975, R-1, p. 43 à 72)

POST GRAMMARS AS A PROGRAMMING LANGUAGE DESCRIPTION TOOL

par Paul FEAUTRIER

Communiqué par G. AUSIELLO

Summary. – *This paper advocates the use of Post grammars for the formal definition of the syntax and semantic of programming languages. As an example, a small Algol-like language is completely specified and the consistency of the resulting formal system is proved.*

I. INTRODUCTION

Post grammars were introduced as formal mathematical objects by Post in [13]. The most accessible description of their properties may be found in Rosenbloom [14]. Post shows in [13] that general Post grammars may be reduced to a somewhat simpler form : the normal grammars. While this is a very important theoretical result, it will not be used here as normal grammars lack the naturalness of general Post grammars.

The aim of this paper is to show how Post grammars may be used to give complete formal definitions of programming languages.

Chapter 2 is of an introductory nature; the definition of Post grammars is stated for reference in the following chapters. No demonstrations are given for the results quoted; these may be found in Rosenbloom [14] or in the author's technical report [4].

As an example of the power of Post grammars, chapter 3 defines the syntax and the semantic of a simple *ad hoc* programming language. The semantical part describes both the flow of control among instructions and the (limited) set of data manipulation statements.

Chapter 4 indicates how to state properties of programs and sketch a proof of such a property in the case of a simple sorting procedure.

Université Pierre et Marie Curie, place Jussieu, Paris.

Chapter 5 compares the present approach with some current work and points to several open problems for future research.

While quite different in aims and underlying concepts, most of the methods and vocabulary of this paper are borrowed from the work of Curry, notably the second chapter of [3] and the third chapter of [2].

2. DEFINITION OF POST GRAMMARS

2.1. Alphabet, substitution, language

Let A be an arbitrary set. As is well known, A^*, the free monoid over A, is the set of all finite strings of elements of A. Concatenation is an associative binary operation on A^* denoted simply by juxtaposition of its operands.

Let $x_1 \ldots x_k$ be letters of A and $a_1 \ldots a_k$ be words of A^*;

The notation :

$$S \begin{matrix} a_1 \ldots a_k \\ x_1 \ldots x_k \end{matrix}$$

represents the operator of simultaneous substitution of the words $a_1 \ldots a_k$ for the letters $x_1 \ldots x_k$ in any word of A^*.

A language is a subset of A^*; a grammar is a prescription for the construction of a language.

2.2. Post productions, Post grammars

Let V be another set; A and V will be supposed distinct. In the sequel, elements of V will always be denoted by small Greek letters and may be called syntactical variables.

A Post production is represented in the form of an inference figure :

$$[\pi] \frac{a_1 \ldots a_k}{c}$$

in which the a_i and c are words over $\{ A \cup V \}$. The a_i are the premises of π; c is its conclusion. It is customary to restrict the syntactical variables in c to the set of those already present in the a_i.

A Post grammar Π is a finite set of Post productions. The corresponding language is defined in the following way :

It is possible for a Post production to have no premises; the production is then called an axiom. It is easy to see that the conclusion of an axiom contains no syntactical variables.

Let $m_1 \ldots m_k, m$ be $k + 1$ words on A. The word m is immediately deducible form $m_1 \ldots m_k$ according to the rule π if there exists a substitution operator S acting only on syntactical variables and such that :

$$S(a_i) = m_i$$
$$S(c) = m.$$

A sequence of words

$$m_1 \ldots m_k$$

is a proof according to Π if each word m_i is immediately deducible from words occuring earlier in the sequence according to some production of Π.

The Post language defined by Π is then simply the set of those words which occur in some proof according to Π.

Any proof may be displayed in tree form. A node of the tree is labeled by a word of the proof and is linked upward to all nodes which were used in its deduction. The construction of the proof tree may involve the duplication of a common subproof.

2.3. Canonical relations

Let r be a word on $A \cup V$. Let $\alpha_1, \ldots, \alpha_k$ be the syntactical variables occuring in r. Let $\mathfrak{L}$ be a language. $\mathfrak{L}$ and r together define a relation on $[A^*]^k$ thus :

$$\{ \mathfrak{L} \vdash r \} (m_1 \ldots m_k) \rightleftarrows S_{\alpha_1}^{m_1} \cdots {}_{\alpha_k}^{m_k}(r) \in L.$$

(this definition is a slight generalisation of the one given by Rosenbloom in [14].)

Any language $\mathfrak{L}$ is identical with its canonical relation $\{ \mathfrak{L} \vdash \alpha \}$. It is interesting to note that for Post languages the converse property is false. (See [4].)

3. FORMAL DEFINITION OF AN ALGOL-LIKE LANGUAGE

As an example of a formal definition by a Post grammar, this chapter describes an *ad hoc* programming language which will be called μ Algol. Note that no claims are made as to the originality or general interest of μ Algol.

3.1. Outline

The formal description of the syntax of a programming language using a formal grammar is quite natural. Let $\mathcal{A}$ be the Post language used. A string α will be a correct programm if the word :

$$\textbf{program}\ [\alpha]$$

is in $\mathcal{A}$. The definition of the meaning of α is, however, more difficult.

One way is to define (informally) an object computer (generally a very simple one) and to give rules for computing the translation of any correct program into the language of the object computer. This may be done with the help of a canonical relation

$$\textbf{compile}\ [\alpha, \beta].$$

This method or variations thereof has been widely used (see for instance Nolin [12] or the work of Knuth [8]). The " compile " relation in fact defines a compiler for the language; this fact is very interesting from the practical point of view, but is more of a nuisance when one tries to derive properties for life-size languages.

In the above approach, program α acts on unspecified objects whose definition is buried in the description of the object computer. One may also use for those objects well choosen strings on the alphabet of $\mathcal{A}$, by providing canonical predicates :

$$\{\ \mathcal{A} \vdash \textbf{integer}\ [\alpha]\ \},$$
$$\{\ \mathcal{A} \vdash \textbf{boolean}\ [\alpha]\ \}\ \text{... as needed.}$$

However, a normal program acts not on an isolated datum, but rather on a data structure : the language $\mathcal{A}$ must provide a predicate

$$\{\ \mathcal{A} \vdash \textbf{structure}\ [\alpha]\ \}$$

with ways and means to build structures from elementary data. The meaning of a program is then a function from data structures to data structures; this may be done by providing a three places relation

$$\textbf{apply}\ [\alpha, \beta, \gamma]$$

with the interpretation that α is a program and β is a data structure and γ is the result of applying α to β. One may say that, while the preceding method did define a language by its compiler, this one defines it by an interpreter.

As a matter of technical convenience, it is possible to formulate the **apply** relation as a reduction rule :

$$\alpha\ \{\ \beta\ \} \rightarrow \nu,$$

the sign $\rightarrow$ being read " yields ". Data structures will be irreducible strings for the $\rightarrow$ operation. The theory will also contain data structure expressions

or constructs. Some of these constructs will be reducible to a data structure which will be the value of the construct. For this system to be consistent, one should prove the uniqueness of this value.

Some constructs will have no value, either because the reduction algorithm does not terminate, or because it terminates in a construct which is not a data structure. These constructs will correspond to semantically uncorrect programs or data.

A data structure is, basically, a partial application of a name space into the elementary data space. However, the data space is, in general, split into many different subspaces or types. The type associated to a particular name is given in a declaration and used when computing the value of expressions. One may either insert the type information in the data structure, or else furnish the current set of declarations as a second argument when evaluating expressions. The second method is simpler as long as the set of valid declaration may be inferred in a static way from the program text. However, this is no longer possible when declarations are interpreted as data structure constructors. The first method will be preferred in this paper.

To help in the construction of large programs, it is customary to break the text in many named procedures, and to invoke them simply by stating their names. It is obvious that such a procedure call cannot be interpreted without some acces to the text of the procedure body. This leads us to use reduction rules of the form :

$$\chi \{ \pi, \sigma \} \rightarrow \tau$$

where χ is a command (declaration, statement or procedure call), π is a program, σ is a data structure and τ is the resultant construct. For all commands except procedure calls, π will be a parametric argument, which is simply transmitted to τ.

To transmit data to and from procedures, conventional programming languages use either parameters or global variables. Furthermore, steps are taken to insure that names are unrelated inside and outside procedures.

The simplest of the mecanisms for the transmission of parameters is the one which uses global variables. For the sake of simplicity this will be the only one defined in μ Algol. One may note that the system of indirect data reference of μ Algol allows one to implement a classical parameter system from inside the language itself.

3.2. Technical preliminaries

3.21. Reducibility

The reduction operator will act on constructs of programs, data structures and data operators to form other constructs. These objects will be described by the canonical predicate **construct** [α] whose definition is given by rules [3]-[5].

Reduction will be a quasi-ordering among constructs :

$$[1] \quad \frac{\alpha \to \beta \quad \beta \to \gamma}{\alpha \to \gamma}$$

$$[2] \quad \frac{\textbf{construct}\ [\alpha]}{\alpha \to \alpha}$$

If a stepwise reduction of a construct is to be possible, all of the construction operators (or constructors) should be monotone with respect to reduction. This will be provided by the following rules.

$$[3] \quad \frac{\textbf{construct}\ [\alpha]}{\textbf{constructlist}\ [\alpha]}$$

$$[4] \quad \frac{\textbf{constructlist}\ [\alpha] \quad \textbf{construct}\ [\beta]}{\textbf{constructlist}\ [\alpha, \beta]}$$

$$[5] \quad \frac{\textbf{constructor}\ [\chi] \quad \textbf{constructlist}\ [\lambda]}{\textbf{construct}\ [\chi \{ \lambda \}]}$$

$$[6] \quad \frac{\alpha \to \beta \quad \gamma \to \delta \quad \textbf{construct}\ [\gamma]}{\alpha, \gamma \to \beta, \delta}$$

$$[7] \quad \frac{\textbf{constructor}\ [\chi] \quad \alpha \to \beta}{\chi \{ \alpha \} \to \chi \{ \beta \}}$$

$$[8] \quad \frac{\textbf{constructor}\ [\chi] \quad \alpha \to \beta \quad \chi \{ \alpha \}}{\chi \{ \beta \}}$$

To the reduction quasi-ordering is associated an equality which is defined by the rules :

$$[9] \quad \frac{\alpha \to \beta}{\alpha = \beta}$$

$$[10] \quad \frac{\alpha \to \beta}{\beta = \alpha}$$

$$[11] \quad \frac{\alpha = \beta \quad \beta = \gamma}{\alpha = \gamma}$$

To shorten the following exposition, it will be stipulated that any function or relation which is constructed with the signs { and } will be monotone and that an axiom to that effect should be added to the language. For instance, one should add to the rules in paragraph 3.3 the following axioms :

$$\frac{}{\textbf{constructor [length]}} \qquad \frac{}{\textbf{constructor [value]}}$$

A list of these axioms is given in appendix *A*.

3.22. Elementary data

To simplify the description of μ Algol, we will use mutually disjoint sets of elementary data. This implies that it will be possible to deduce the type of a datum from the datum itself.

The construction of types is a rather straightforward and uninteresting matter; to shorten the exposition, only very sketchy indications will be given here. For more detailed examples, the reader may consult [4].

A type is defined by a canonical predicate :

$$\textbf{integer } [\alpha]$$

for instance. Operators on elementary data are defined by reductions rules with conclusions of the form :

$$\omega \{ \alpha_1 \ldots \alpha_n \} \rightarrow \beta$$

where it is understood that ω is a constructor and that the α_i and β need not all be of the same type.

It will be postulated that elementary data are not reducible to each others and that the defining rules for an operator satisfy the conditions for a proper definitional extension (cf. Curry and Feys [3] par. 2E). These conditions insure the consistency of the construction.

All types are assumed to contain the indefinite datum ∞ ; a construct containing ∞ will be irreducible.

The set of types is defined by the following axioms :

$$[1] \ \frac{}{\textbf{type [integer]}}$$

$$[2] \ \frac{}{\textbf{type [boolean]}}$$

$$[3] \ \frac{}{\textbf{type [string]}}$$

3.221. Integers

Integers are strings on the alphabet 0, 1, 2, ..., 9, optionally preceded by the negative sign, −. The operators are the four usual arithmetic operations, the remainder operation, and the comparison operators, whose results are of type boolean.

3.222. Boolean values

The two boolean values are **true** and **false**. The operators on these are the usual boolean connectives, **and**, **or**, **not**, etc.

3.223. Strings

A string is a word on a given alphabet, (e.g. *a*-*z*, 0-9, etc) enclosed between quote marks : " − ".

Operations on strings are :

− concatenation, defined by :

$$[1]\quad \frac{\textbf{string}\,[\,"\alpha"\,]\quad \textbf{string}\,[\,"\beta"\,]}{|\,\{\,"\alpha",\,"\beta"\,\}\rightarrow "\alpha\beta"};$$

− comparison operators, with boolean results ;

− conversion operators, from strings to integers and conversely.

The word of zero length is explicitly included among the strings.

The syntactical predicate **diff** [α, β] will be asserted if the strings α and β are note equal.

3.3. Data structures

An identifier is a string striped of its enclosing quotes :

$$[1]\quad \frac{\textbf{string}\,[\,"\alpha"\,]}{\textbf{identifier}\,[\alpha]}.$$

A τ **vector** is a string of data of type τ, separated by spaces (␣). Similarly, an ∞ **vector** is a string containing only ∞ signs. Note however that ∞ is not a type. To each string is associated its length.

$$[2]\quad \frac{\textbf{type}\,[\tau]}{\tau\ \textbf{vector}\,[\ ]}$$

$$[3]\quad \frac{}{\infty\ \textbf{vector}\,[\ ]}$$

$$[4]\quad \frac{}{\infty\,[\infty]}$$

$$[5]\ \frac{\tau\ \textbf{vector}\ [\sigma]\ \tau[\varepsilon]}{\tau\ \textbf{vector}\ [\sigma \sqcup \varepsilon]}$$

$$[6]\ \frac{}{\textbf{length}\ \{\}\ \rightarrow\ 0}$$

$$[7]\ \frac{\tau\ \textbf{vector}\ [\sigma]\quad \tau[\varepsilon]}{\textbf{length}\ \{\sigma \sqcup \varepsilon\}\ \rightarrow\ +\ \{\textbf{length}\ \{\sigma\},\ 1\}}$$

The **index** and **set** functions enable one to consult and modify a given element in a string.

$$[8]\ \frac{\tau\ \textbf{vector}\ [\delta]\quad \tau\ \textbf{vector}\ [\alpha]\quad \tau\ \textbf{vector}\ [\phi]\quad \textbf{length}\ \{\delta\}\ \rightarrow\ \nu\quad \textbf{length}\ \{\alpha\}\ \rightarrow\ \lambda}{\textbf{index}\ \{\delta\alpha\phi,\ \nu,\ \lambda\}\ \rightarrow\ \alpha}$$

$$[9]\ \frac{\tau\ \textbf{vector}\ [\delta]\quad \textbf{length}\ \{\delta\}\ \rightarrow\ \nu\quad \tau\ \textbf{vector}\ [\phi]\quad \tau\ \textbf{vector}\ [\alpha]\quad \textbf{length}\ \{\alpha\}\ \rightarrow\ \lambda\quad \tau\ \textbf{vector}\ [\beta]}{\textbf{set}\ \{\delta\alpha\phi,\ \beta,\ \nu,\ \lambda\}\ \rightarrow\ \delta\beta\phi}$$

Remark that indexes start at zero and that the replacing vector, β, is not necessarily of the same length as the replaced one.

3.31. Data entry

A data entry associates a type and a string of values to an identifier. Those informations may be extracted from the data entry with the help of the **tag** and **value** functions. The result of these functions is indefinite (∞) for all identifiers differing from the one used to construct the entry.

$$[1]\ \frac{\textbf{identifier}\ [\nu]\quad \tau\ \textbf{vector}\ [\sigma]}{\textbf{entry}\ [\nu\tau(\sigma)]}$$

$$[2]\ \frac{\textbf{identifier}\ [\nu]\quad \tau\ \textbf{vector}\ [\sigma]}{\textbf{value}\ \{\nu\tau(\sigma),\ ''\ \nu\ ''\}\ \rightarrow\ \sigma}$$

3.32. Data block

A data block is a string of data entries with the restriction that no identifier is used more than once. There is a natural extension of the **tag** and **value** functions to data blocks. A block may be empty.

$$[1]\ \frac{}{\textbf{block}\ [\ ]}$$

$$[2]\ \frac{\textbf{block}\ [\beta]\quad \textbf{identifier}\ [\nu]\quad \textbf{entry}\ [\nu\rho]\quad \textbf{tag}\ [\beta,\ ''\ \nu\ '',\ \infty]}{\textbf{block}\ [\beta\nu\rho]}$$

[3] $\dfrac{\textbf{identifier}\ [\nu]}{\textbf{tag}\ [\ ,\ "\ \nu\ ",\ \infty]}$

[4] $\dfrac{\textbf{block}\ [\beta\nu\tau(\omega)]\quad \textbf{identifier}\ [\nu]\quad \textbf{type}\ [\tau]\quad \tau\ \textbf{vector}\ [\omega]}{\textbf{tag}\ [\beta\nu\tau(\omega),\ "\ \nu\ ",\ \tau]}$

[5] $\dfrac{\begin{array}{c}\textbf{block}\ [\beta\nu\rho]\quad \textbf{identifier}\ [\nu]\quad \textbf{entry}\ [\nu\rho]\\ \textbf{tag}\ [\beta,\ "\ \mu\ ",\ \tau]\quad \textbf{diff}\ ["\ \mu\ ",\ "\ \nu\ "]\end{array}}{\textbf{tag}\ [\beta\nu\rho,\ "\ \mu\ ",\ \tau]}$

[6] $\dfrac{\textbf{identifier}\ [\nu]\quad \textbf{integer}\ [\lambda]}{\textbf{value}\ \{\ ,\ "\ \nu\ ",\ \lambda\ \} \rightarrow \infty}$

[7] $\dfrac{\textbf{block}\ [\beta\nu\rho]\quad \textbf{identifier}\ [\nu]\quad \textbf{entry}\ [\nu\rho]\quad \textbf{integer}\ [\lambda]}{\textbf{value}\ \{\ \beta\nu\rho,\ "\ \nu\ ",\ \lambda\ \} \rightarrow \textbf{value}\ \{\ \nu\rho,\ "\ \nu\ ",\ \lambda\ \}}$

[8] $\dfrac{\begin{array}{c}\textbf{block}\ [\beta\nu\rho]\quad \textbf{identifier}\ [\nu]\quad \textbf{entry}\ [\nu\rho]\quad \textbf{integer}\ [\lambda]\\ \textbf{diff}\ ["\ \mu\ ",\ "\ \nu\ "]\quad \textbf{identifer}\ [\mu]\end{array}}{\textbf{value}\ \{\ \beta\nu\rho,\ "\ \mu\ ",\ \lambda\ \} \rightarrow \textbf{value}\ \{\ \beta,\ "\ \mu\ ",\ \lambda\ \}}$

3.33. Data structure

A data structure is a string of blocks separated by the $\not\equiv$ sign. Each block is created by a new procedure call and introduces an independant system of names. The informations returned by the tag and values functions are extracted from the rightmost block in the structure in which the identifier is defined. This provide a method for transmitting information to and from a procedure.

[1] $\dfrac{\textbf{block}\ [\beta]}{\textbf{structure}\ [\beta]}$

[2] $\dfrac{\textbf{structure}\ [\sigma]\quad \textbf{block}\ [\beta]}{\textbf{structure}\ [\sigma \not\equiv \beta]}$

[3] $\dfrac{\textbf{structure}\ [\sigma]\quad \textbf{block}\ [\beta]\quad \textbf{tag}\ [\beta, \nu, \tau]\quad \textbf{type}\ [\tau]}{\textbf{tag}\ [\sigma \not\equiv \beta, \nu, \tau]}$

[4] $\dfrac{\textbf{structure}\ [\sigma]\quad \textbf{block}\ [\beta]\quad \textbf{tag}\ [\beta, \nu, \infty]\quad \textbf{tag}\ [\sigma, \nu, \tau]}{\textbf{tag}\ [\sigma \not\equiv \beta, \nu, \tau]}$

[5] $\dfrac{\textbf{structure}\ [\sigma]\quad \textbf{block}\ [\beta]\quad \textbf{tag}\ [\beta, \nu, \tau]\quad \textbf{type}\ [\tau]\quad \textbf{integer}\ [\lambda]}{\textbf{value}\ \{\ \sigma \not\equiv \beta, \nu, \lambda\ \} \rightarrow \textbf{value}\ \{\ \beta, \nu, \lambda\ \}}$

[6] $\dfrac{\textbf{structure}\ [\sigma]\quad \textbf{block}\ [\beta]\quad \textbf{tag}\ [\beta, \nu, \infty]\quad \textbf{integer}\ [\lambda]}{\textbf{value}\ \{\ \sigma \not\equiv \beta, \nu, \lambda\ \} \rightarrow \textbf{value}\ \{\ \sigma, \nu, \lambda\ \}}$

3.4. Statements

A statement is the name of a function from data structures to data structures. An expression is a common part of many statements and is defined

separately to avoid repetitions. Statements are divided in declarations, which construct data structures and commands, which modify them.

Recalling the discussion at the beginning of paragraph 3, a statement will be defined by reduction rules of the form :

$$\chi \{ \pi, \sigma \} \rightarrow \tau$$

In all cases except in the procedure call command, the π argument will simply be a parameter which is carried along in the reduction.

3.41. Expressions

An expression is the name of a function from data structures to elementary data. Expressions occur as components in various statements.

The primary components of expressions are constants and (sometime qualified) identifiers. From those primaries one builds more complicated expressions with the help of operators. A constant stands for itself. An identifier stands for the value which is associated to it in the data structure. From a qualified identifier one builds another identifier by concatenation of the value of the qualifier and the qualified name.

An operator has a priority and a multiplicity (a number of arguments). All operators will have a multiplicity of 1 or 2. These quantities will be represented by strings of ′ marks in the required number. To each operator will be attached defining axioms in the form :

$$\overline{\mu - \nu \textbf{ operator } [\mathrm{Op}]}$$

where μ is the (′ string representing the) multiplicity and ν is the priority. For instance, the description of the boolean complement is :

$$\overline{'' - '''''' \textbf{ operator } [\textbf{not}]}$$

while the equality operator has the description :

$$\overline{'' - '''''' \textbf{ operator } [\textbf{eq}]}$$

Appendix *B* gives a list of defining axioms for the operators of μ Algol.

The canonical class ν **term** $[\alpha]$ where ν is a ′ string will represent expressions constructed from primaries and operators of priority no higher than ν. When ν is the null string, α will simply be a primary.

3.411. Primaries

There are three kinds of primaries :

– constants

$$[1]\quad \frac{\textbf{type}\ [\tau]\quad \tau[\chi]}{\textbf{term}\ [\chi]}$$

$$[2]\quad \frac{\textbf{type}\ [\tau]\quad \tau[\chi]\quad \textbf{structure}\ [\sigma]}{\chi\ \{\ \sigma\ \}\ \rightarrow\ \chi}$$

– identifiers (possibly indexed and qualified)

$$[3]\quad \frac{\textbf{identifier}\ [\nu]}{\textbf{ref}\ [\nu]}$$

$$[4]\quad \frac{\textbf{ref}\ [\rho]\quad \textbf{identifier}\ [\nu]}{\textbf{ref}\ [\rho\ .\ \nu]}$$

$$[5]\quad \frac{\textbf{ref}\ [\rho]\quad \textbf{identifier}\ [\nu]\quad \textbf{expression}\ [\varepsilon]}{\textbf{ref}\ [\rho(\varepsilon)\ .\ \nu]}$$

$$[6]\quad \frac{\textbf{identifier}\ [\nu]\quad \textbf{structure}\ [\sigma]}{\textbf{name}\ \{\ \sigma,\ \nu\ \}\ \rightarrow\ ''\ \nu\ ''}$$

$$[7]\quad \frac{\textbf{ref}\ [\rho]\quad \textbf{identifier}\ [\nu]\quad \textbf{structure}\ [\sigma]}{\textbf{name}\ \{\ \sigma,\ \rho\ .\ \nu\ \}\ \rightarrow\ |\ \{\ \textbf{index}\ \{\ \textbf{value}\ \{\ \sigma,\ \textbf{name}\ \{\ \sigma,\ \rho\ \}\},\ 0,\ 1\ \},\ ''\ \nu\ ''\ \}}$$

$$[8]\quad \frac{\textbf{ref}\ [\rho]\quad \textbf{identifier}\ [\nu]\quad \textbf{structure}\ [\sigma]\quad \textbf{expression}\ [\varepsilon]}{\textbf{name}\ \{\ \sigma,\ \rho(\varepsilon)\ .\ \nu\ \}\ \rightarrow\ |\ \{\ \textbf{index}\ \{\ \textbf{value}\ \{\ \sigma,\ \textbf{name}\ \{\ \sigma,\ \rho\ \}\},\ \varepsilon\ \{\ \sigma\ \},\ 1\ \},\ ''\ \nu\ ''\}}$$

$$[9]\quad \frac{\textbf{ref}\ [\rho]}{\textbf{term}\ [\rho]}$$

$$]10]\quad \frac{\textbf{ref}\ [\rho]\quad \textbf{structure}\ [\sigma]}{\rho\ \{\ \sigma\ \}\ \rightarrow\ \textbf{index}\ \{\ \textbf{value}\ \{\ \sigma,\ \textbf{name}\ \{\ \sigma,\ \rho\ \}\},\ 0,\ 1\ \}}$$

$$[11]\quad \frac{\textbf{ref}\ [\rho]\quad \textbf{expression}\ [\varepsilon]}{\textbf{term}\ [\rho(\varepsilon)]}$$

$$[12]\quad \frac{\textbf{ref}\ [\rho]\quad \textbf{expression}\ [\varepsilon]\quad \textbf{structure}\ [\sigma]}{\rho(\varepsilon)\ \{\ \sigma\ \}\ \rightarrow\ \textbf{index}\ \{\ \textbf{value}\ \{\ \sigma,\ \textbf{name}\ \{\ \sigma,\ \rho\ \}\},\ \varepsilon\ \{\ \sigma\ \},\ 1\ \}}$$

– expressions enclosed in parenthesis :

$$[13]\quad \frac{\textbf{expression}\ [\varepsilon]}{\textbf{term}\ [(\varepsilon)]}$$

$$[14]\quad \frac{\textbf{expression}\ [\varepsilon]\quad \textbf{structure}\ [\sigma]}{(\varepsilon)\ \{\ \sigma\ \}\ \rightarrow\ \varepsilon\ \{\ \sigma\ \}}$$

3.412. Operators

There are three cases in the definition of higher level terms.

3.4121. Any ν term is also a ν′ term :

$$[1]\ \frac{\nu\ \textbf{term}\ [\alpha]}{\nu'\ \textbf{term}\ [\alpha]}$$

3.4122. A ′ − ν ′ operator acting on a ν term gives a ν′ term :

$$[1]\ \frac{\nu\ \textbf{term}\ [\alpha]\quad '-\nu'\ \textbf{operator}\ [\omega]}{\nu'\ \textbf{term}\ [\omega\alpha]}$$

$$[2]\ \frac{\nu\ \textbf{term}\ [\alpha]\quad '-\nu'\ \textbf{operator}\ [\omega]\quad \textbf{structure}\ [\sigma]}{\omega\alpha\ \{\sigma\} \rightarrow \omega\ \{\alpha\ \{\sigma\}\}}$$

3.4123. A ″ − ν′ operator acting on a ν′ term and a ν term gives another ν′ term. Note that this definition includes the familiar rule of association from the left :

$$[1]\ \frac{\nu'\ \textbf{term}\ [\alpha]\quad ''-\nu'\ \textbf{operator}\ [\omega]\quad \nu\ \textbf{term}\ [\beta]}{\nu'\ \textbf{term}\ [\alpha\omega\beta]}$$

$$[2]\ \frac{\nu'\ \textbf{term}\ [\alpha]\quad ''-\nu'\ \textbf{operator}\ [\omega]\quad \nu\ \textbf{term}\ [\beta]\quad \textbf{structure}\ [\sigma]}{\alpha\omega\beta\ \{\sigma\} \rightarrow \omega\ \{\alpha\ \{\sigma\}, \beta\ \{\sigma\}\}}$$

Any term is an expression :

$$[3]\ \frac{\nu\ \textbf{term}\ [\alpha]}{\textbf{expression}\ [\alpha]}$$

3.42. Declarations

In accordance with dataless programming techniques, declarations are here defined as data structure constructors. The identifier of the new data entry is the current value of the occuring reference. A declaration may occur anywhere in the program text (if the occuring reference and expression are defined).

$$[1]\ \frac{\textbf{type}\ [\tau]\quad \textbf{ref}\ [\rho]}{\textbf{statement}\ [\tau\rho]}$$

$$[2]\ \frac{\textbf{type}\ [\tau]\quad \textbf{ref}\ [\rho]\quad \textbf{expression}\ [\varepsilon]}{\textbf{statement}\ [\tau\rho(\varepsilon)]}$$

$$[3]\ \frac{\begin{array}{c}\textbf{type}\ [\tau]\quad \textbf{program}\ [\pi]\quad \textbf{ref}\ [\rho]\quad \textbf{name}\ \{\sigma, \rho\} \rightarrow ''\nu'' \\ \textbf{structure}\ [\sigma\nu\tau(\sqcup\ \infty)]\end{array}}{\tau\rho\ \{\pi, \sigma\} \rightarrow \sigma\nu\tau(\sqcup\infty)}$$

$$[4]\ \frac{\begin{array}{l}\textbf{type}\ [\tau]\quad \textbf{program}\ [\pi]\quad \textbf{ref}\ [\rho]\quad \textbf{expression}\ [\varepsilon] \\ \textbf{name}\ \{\sigma, \rho\} \rightarrow ''\nu''\quad \infty\ \textbf{vector}\ [\omega]\quad \textbf{structure}\ [\sigma\nu\tau(\omega)] \\ \varepsilon\ \{\sigma\} \rightarrow \lambda\quad \textbf{length}\ \{\omega\} \rightarrow \lambda\quad \textbf{integer}\ [\lambda]\end{array}}{\tau\rho(\varepsilon)\ \{\pi, \sigma\} \rightarrow \sigma\nu\tau(\omega)}$$

3.43. Commands

3.431. The empty command

$$[1]\ \frac{}{\textbf{statement}\ [\ ]}$$

$$[2]\ \frac{\textbf{program}\ [\pi]\quad \textbf{structure}\ [\sigma]}{\{\ \pi, \sigma\ \} \rightarrow \sigma}$$

3.432. The assignment command

$$[1]\ \frac{\textbf{ref}\ [\rho]\quad \textbf{expression}\ [\varepsilon]}{\textbf{statement}\ [\rho : = \varepsilon]}$$

$$[2]\ \frac{\textbf{ref}\ [\rho]\quad \textbf{expression}\ [\varepsilon]\quad \textbf{expression}\ [\lambda]}{\textbf{statement}\ [\rho(\lambda) : = \varepsilon]}$$

$$[3]\ \frac{\textbf{ref}\ [\rho]\quad \textbf{expression}\ [\varepsilon]\quad \textbf{program}\ [\pi]\quad \textbf{structure}\ [\sigma]}{\rho : = \varepsilon\ \{\ \pi, \sigma\ \} \rightarrow \rho(0) : = \varepsilon\ \{\ \pi, \sigma\ \}}$$

$$[4]\ \frac{\begin{array}{c}\textbf{structure}\ [\theta]\quad \textbf{structure}\ [\omega] \\ \textbf{name}\ \{\ \theta\nu\tau(\alpha)\omega, \rho\ \} \rightarrow "\nu" \\ \textbf{tag}\ [\omega, "\nu", \infty]\quad \tau\ \textbf{vector}\ [\beta]\end{array}}{\textbf{modify}\ \{\ \theta\nu\tau(\alpha)\omega, \rho, \beta\ \} \rightarrow \theta\nu\tau(\beta)\omega}$$

In the above rule, one should note that the modified occurence of " ν " is always the rightmost one.

$$[5]\ \frac{\begin{array}{c}\textbf{ref}\ [\rho]\quad \textbf{expression}\ [\varepsilon]\quad \textbf{expression}\ [\lambda] \\ \textbf{program}\ [\pi]\quad \textbf{structure}\ [\sigma]\end{array}}{\rho(\lambda) : = \varepsilon\ \{\ \pi, \sigma\ \} \rightarrow \textbf{modify}\ \{\ \sigma, \rho, \textbf{set}\ \{\ \textbf{value}\ \{\ \sigma, \textbf{name}\ \{\ \sigma, \rho\ \}\}, \varepsilon\ \{\ \sigma\ \}, \lambda\ \{\ \sigma\ \}, 1\ \}\}}$$

3.433. The conditional statement

$$[1]\ \frac{\textbf{expression}\ [\beta]\quad \textbf{body}\ [\tau]\quad \textbf{body}\ [\phi]}{\textbf{statement}\ [\textbf{if}\ \beta\ \textbf{then}\ \tau\ \textbf{else}\ \phi\ \textbf{endif}]}$$

$$[2]\ \frac{\begin{array}{c}\textbf{program}\ [\pi]\quad \textbf{expression}\ [\beta]\quad \textbf{body}\ [\tau]\quad \textbf{body}\ [\phi] \\ \beta\ \{\ \sigma\ \} \rightarrow \textbf{true}\end{array}}{\textbf{if}\ \beta\ \textbf{then}\ \tau\ \textbf{else}\ \phi\ \textbf{endif}\ \{\ \pi, \sigma\ \} \rightarrow \tau\ \{\ \pi, \sigma\ \}}$$

$$[3]\ \frac{\begin{array}{c}\textbf{program}\ [\pi]\quad \textbf{expression}\ [\beta]\quad \textbf{body}\ [\tau]\quad \textbf{body}\ [\phi] \\ \beta\ \{\ \sigma\ \} \rightarrow \textbf{false}\end{array}}{\textbf{if}\ \beta\ \textbf{then}\ \tau\ \textbf{else}\ \phi\ \textbf{endif}\ \{\ \pi, \sigma\ \} \rightarrow \phi\ \{\ \pi, \sigma\ \}}$$

The definition of a (procedure) body is postponed until paragraph 3.435.

3.434. The iteration statement

$$[1]\quad \frac{\textbf{expression}\ [\beta]\quad \textbf{body}\ [\rho]}{\textbf{statement}\ [\textbf{while}\ \beta\ \textbf{do}\ \rho\ \textbf{endwhile}]}$$

$$[2]\quad \frac{\textbf{program}\ [\pi]\quad \textbf{structure}\ [\sigma]\quad \textbf{body}\ [\rho]\quad \beta\ \{\ \sigma\ \} \rightarrow \textbf{true}}{\textbf{while}\ \beta\ \textbf{do}\ \rho\ \textbf{endwhile}\ \{\ \pi, \sigma\ \} \rightarrow \textbf{while}\ \beta\ \textbf{do}\ \rho\ \textbf{endwhile}\ \{\ \pi, \rho\ \{\ \pi, \sigma\ \}\}}$$

$$[3]\quad \frac{\textbf{program}\ [\pi]\quad \textbf{structure}\ [\sigma]\quad \textbf{body}\ [\rho]\quad \beta\ \{\ \sigma\ \} \rightarrow \textbf{false}}{\textbf{while}\ \beta\ \textbf{do}\ \rho\ \textbf{endwhile}\ \{\ \pi, \sigma\ \} \rightarrow \sigma}$$

3.435. The procedure body

A procedure body is a string of statements. The sequential execution of these statements correspond to the composition of the associated semantical functions.

$$[1]\quad \frac{\textbf{statement}\ [\chi]}{\textbf{body}\ [\chi]}$$

$$[2]\quad \frac{\textbf{body}\ [\beta]\quad \textbf{statement}\ [\chi]}{\textbf{body}\ [\chi;\ \beta]}$$

$$[3]\quad \frac{\textbf{program}\ [\pi]\quad \textbf{body}\ [\beta]\quad \textbf{statement}\ [\chi]\quad \textbf{structure}\ [\sigma]}{\chi;\ \beta\ \{\ \pi, \sigma\ \} \rightarrow \beta\ \{\ \pi, \chi\ \{\ \pi, \sigma\ \}\}}$$

3.436. The procedure call statement

In μ Algol, a procedure has no explicit arguments; communication is achieved only through common identifiers. However, one may reconstruct a parameter mecanism with the help of qualified identifiers. To allow the redefinition of identifiers within a procedure body, the input structure is marked at the moment of the call with the $\not\equiv$ delimiter. When the execution of the procedure body is completed, the $\not\equiv$ sign and all information to the right of it are discarded.

A program is a string of named procedure bodies. No identifier should occur more than once as a procedure name in a program. This condition is formalized with the help of the auxiliary predicate **notin**.

$$[1]\quad \frac{}{\textbf{program}\ [\ ]}$$

$$[2]\quad \frac{\textbf{program}\ [\pi]\quad \textbf{identifier}\ [\nu]\quad \nu\ \textbf{notin}\ [\pi]\quad \textbf{body}\ [\beta]}{\textbf{program}\ [\pi\ \textbf{procedure}\ \nu\ \textbf{begin}\ \beta\ \textbf{end}]}$$

$$[3]\quad \frac{\textbf{identifier}\ [\nu]}{\nu\ \textbf{notin}\ [\ ]}$$

$$[4]\quad \frac{\textbf{program}\ [\pi]\quad \textbf{identifier}\ [\nu]\quad \nu\ \textbf{notin}\ [\pi]\quad \textbf{body}\ [\beta]\quad \mu\ \textbf{notin}\ \pi\quad \textbf{diff}\ [''\mu'', ''\nu'']}{\mu\ \textbf{notin}\ [\pi\ \textbf{procedure}\ \nu\ \textbf{begin}\ \beta\ \textbf{end}]}$$

$$[5]\quad \frac{\textbf{identifier}\ [\nu]}{\textbf{statement}\ [\textbf{call}\ \nu]}$$

$$[6]\quad \frac{\textbf{program}\ [\theta\ \textbf{procedure}\ \nu\ \textbf{begin}\ \beta\ \textbf{end}\ \omega]\quad \textbf{identifier}\ [\nu]\quad \textbf{body}\ [\beta]\quad \textbf{structure}\ [\sigma]\quad \beta\ \{\ \theta\ \textbf{procedure}\ \nu\ \textbf{begin}\ \beta\ \textbf{end}\ \omega, \sigma \not\equiv\ \} \rightarrow \tau \not\equiv \rho\quad \textbf{block}\ [\rho]}{\textbf{call}\ \nu\ \{\ \theta\ \textbf{procedure}\ \nu\ \textbf{begin}\ \beta\ \textbf{end}\ \omega, \sigma\ \} \rightarrow \tau}$$

3.5. The consistency proof

If the Post grammar $\mathcal{A}$ of paragraphs 3.134 really describes a programming language, there should exist an algorithm to find the result of applying a program to a data structure, that is to say for the reduction of constructs of the form :

$$\textbf{call}\ \nu\ \{\ \pi, \sigma\ \}.$$

Furthermore, if the above construct may be reduced to a data structure, then this structure should be unique. In the following a construct will be " elementary " if it is an elementary datum or a data structure or a reference. Elementary constructs clearly are irreducible. The above properties are guaranteed by the

Theorem 1. If χ is a construct with the reduction :

$$\chi \rightarrow \sigma$$

where σ is elementary, then σ is unique and can be found by an effective process.

Here the phrase " χ is a construct " will be taken as implying that a proof in $\mathcal{A}$ of the word **construct** $[\chi]$ is available.

The rules of $\mathcal{A}$ may be classified into four categories :

– the ancillary rules are those whose conclusion is not a reduction formula;

– the special rules are the rules in paragraphs 3.22 to 3.4;

– the rules of monotony are 3.21 [6] and [7];

– the rule of transitivity is rule 3.21 [1].

It is easy to convince oneself by inspection of the following facts about $\mathcal{A}$:

– The ancillary rules are self-contained in the sense that none of their premises are reduction formulae;

– the conclusion of all special rules is of the form :

$$\omega \{ \alpha_1, \alpha_2 \ldots \alpha_n \} \rightarrow \beta$$

where ω is a constructor and the α_i are elementary constructs.

Lemma. If χ is a construct of the form

$$\omega \{ \alpha_1 \ldots \alpha_n \}$$

where ω is a constructor and where all the α_i are elementary, then there exists a finite set of constructs $\chi_1 \ldots \chi_m$ (the subordinate constructs of χ) such that :

– the χ_i are uniquely determined by χ;

– no reduction exists for χ unless all the subordinate constructs are reducible to elementary constructs;

– there is at most one reduction rule applicable to χ and this rule and its result are uniquely determined when the elementary reductions of all χ_i are known.

The proof of this lemma proceeds by exhausting all the possible cases and is left to the reader. As an example, however, let us consider the case in which ω begins by the letter **if**. ω then has a unique representation in the form :

$$\omega \equiv \textbf{if } \beta \textbf{ then } \tau \textbf{ else } \phi \textbf{ endif}$$

The subordinate construct is $\beta \{ \sigma \}$. The only rules to be applied for the reduction of χ are 3.433 [2] and [3]. These rules contain the premises $\beta \{ \sigma \} \rightarrow$ **true** and $\beta \{ \sigma \} \rightarrow$ **false** and hence cannot be applied unless $\beta \{ \sigma \}$ has an elementary reduction. If $\beta \{ \sigma \}$ reduce to **true**, the rule to be used is 3.433 [2]; if $\beta \{ \sigma \}$ reduce to **false**, the rule to be used is 3.433 [3]; if $\beta \{ \sigma \}$ reduce to any other elementary construct, then χ cannot be further reduced.

A normal proof for a reduction formula is a proof which satisfies the following two conditions :

– if rule 3.21 [7] is applied, then the members of the construction β are all elementary;

– if rule 3.21 [1] is applied, then its second premise, $\beta \rightarrow \gamma$ is not the conclusion of another application of the same rule.

From any proof of a reduction, it is possible to build a normal proof for the same reduction. The first stage of the process eliminates all abnormal uses of rule 3.21 [7] :

$$\frac{\textbf{constructor } [\chi] \quad \alpha \rightarrow \beta}{\chi \{ \alpha \} \rightarrow \chi \{ \beta \}}$$

If some member of β is not elementary, then according to the above discussion, no special rule is applicable to the reduction of $\chi \{ \beta \}$, The proof

tree must contain another application of 3.21 [7], followed by an application of 3.21 [1] :

$$\frac{\dfrac{\textbf{constructor}\ [\chi] \quad \alpha \to \beta}{\chi\{\alpha\} \to \chi\{\beta\}} \qquad \dfrac{\textbf{constructor}\ [\chi] \quad \beta \to \gamma}{\chi\{\beta\} \to \chi\{\gamma\}}}{\chi\{\alpha\} \to \chi\{\gamma\}}$$

This may be rearranged thus :

$$\frac{\textbf{constructor}\ [\chi] \quad \dfrac{\alpha \to \beta \quad \beta \to \gamma}{\alpha \to \gamma}}{\omega\{\alpha\} \to \omega\{\gamma\}}$$

If some member of γ is still not elementary, the process may be continued, going downward in the proof tree. The process eventually stops, as the proof tree is finite ; if in the last modified node γ is not elementary, then no reduction rules are applicable to it ; it is either the bottom node or a superfluous part of the tree and can be eliminated altogether.

The above process may be applied to all abnormal instances of the rule 3.21 [7], in a top-down order, until all of them are eliminated.

Similarly, let us consider an abnormal instance of 3.21 [1].

$$\frac{\alpha \to \beta \quad \dfrac{\beta \to \gamma \quad \gamma \to \delta}{\beta \to \delta}}{\alpha \to \delta}$$

One may rewrite it thus :

$$\frac{\dfrac{\alpha \to \beta \quad \beta \to \gamma}{\alpha \to \gamma} \quad \gamma \to \delta}{\alpha \to \delta}$$

If $\beta \to \gamma$ still is the conclusion of an instance of 3.21 [1], then the process may be iterated going upward in the tree ; the iteration must eventually stop when one reaches a leaf of the proof tree.

The above process is clearly seen to reduce by one the number of abnormal uses of 3.21 [1], and may be applied in bottom-up order, until all of them are eliminated ; the resultant proof tree is then normal. Theorem I then follows by an induction on the depth of the normal proof tree of $\chi \to \sigma$.

In the case of a one step proof, the bottom node is necessarily a reduction by a special rule, and the unicity of σ follows from the lemma.

Let the theorem be true for all proofs with at most n levels, and let $\nu \to \sigma$ have a proof with $n + 1$ levels. The bottom node is an instance of a special rule or of 3.21 [1].

In the first case, the subordinate constructs of χ have proofs with at most n levels and hence, by the induction hypothesis, their elementary reductions are uniquely defined. It then follow from the lemma that the elementary reduction of χ is also unique.

In the second case, as the proof is normal, there is a string of constructs $\chi_o \dots \chi_n$ such that χ_o is χ, χ_n is σ, $\chi_{i-1} \rightarrow \chi_i$ is a word in $\mathcal{A}$ and is not the conclusion of another instance of 3.21 [1].

If one of these partial reductions is the conclusion of a special rule, then one may show with the help of the lemma that its right hand side is uniquely determined by its left hand side; if it is the conclusion of rule 3.21 [7], then, as the proof is normal, χ_{i-1} is of the form $\omega \{ \alpha_1 \dots \alpha_p \}$; the premises of 3.21 [7] are elementary reductions of at most n levels and hence are uniquely defined.

This implies that each construct in the chain χ, $\chi_1 \dots \chi_{n-1}$, is uniquely defined by the preceding one; as the chain must stop when an elementary construct is found, this construct is uniquely determined by the first element of the chain. This completes the proof of Theorem 1. The set of formulas $\chi \rightarrow \chi_1$, $\chi_1 \rightarrow \dots \chi_{n-1} \rightarrow \sigma$ in the normal proof of $\chi \rightarrow \sigma$ will be called its principal branch. From the above discussion it is possible to deduce the

Theorem II. If $\chi \rightarrow \chi'$, then either χ' is χ or χ' is on the principal branch of the normal proof for χ.

To prove theorem II, consider the principal branch of the normal proof of $\chi \rightarrow \chi'$. According to the discussion in the proof of theorem I, the steps of a principal branch beginning with χ are uniquely defined. Hence the principal branch of $\chi \rightarrow \chi'$ is an initial segment of the principal branch of $\chi \rightarrow \sigma$. QED.

From this theorem follows a kind of Church-Rosser property :

Theorem III. If $\chi \rightarrow \sigma$ where σ is elementary, and if $\chi = \chi'$, then $\chi' \rightarrow \sigma$.

$\chi = \chi'$ is the conclusion of one of the three rules 3.21 [9], [10] or [11]. In the first case, theorem III follows by an application of 3.21 [1]. In the second case, theorem III follows from theorem II. In the last case, theorem III is proved by an induction on the number of applications of rule 3.21 [11].

From theorem III it follows that equality is an equivalence relation on the set of constructs with elementary reductions. The quotient set is a model for all constructors in $\mathcal{A}$ according to rule 3.21 [7] and [8]; this model is not trivial as there is no reduction from an elementary construct to another one. Hence $\mathcal{A}$ is consistent.

4. PROPERTIES OF ALGORITHMS

In the light of the discussion of the preceding paragraph, one may now see in what sense the semantic of a program is defined by the grammar $\mathcal{A}$.

The primitive functions of the system are the substitution functions as defined in paragraph 2.11. For instance, an unary function may be defined by a word F on $A \cup \{ \alpha \}$ where α is an arbitrary element of V. To any word a of A^*, this function associates the word $S_\alpha^a(F)$. This definition may be extended to n-ary functions.

The rules of $\mathcal{A}$ define a subset of A^*, the constructs, and an equivalence relation on it. Some equivalence classes contain an irreducible construct, and there is an algorithm for finding this construct if it exists.

Depending on the word F, some substitution functions are monotone with respect to the equivalence relation between constructs. Such a function may be considered as defining an application on the equivalence classes, or on the irreductible constructs which represent them. The semantic of a program is specified by associating to it one of those monotone substitution functions.

It is natural to use relations between the data and results of a program to specify properties of this program. In the present framework, a canonical binary relation may be defined by an extension of $\mathcal{A}$ and a word containing two syntactical variables. Such a relation may be interpreted as a program property if it is monotone; in accordance with the conventions set up in paragraph 3.21, it is then written in the form $r \{ \alpha, \beta \}$.

Let $\mathcal{A}'$ be an extension of $\mathcal{A}$ in which a property r is defined. The procedure p of program $\mathcal{P}$ has property r if the fact that

$$\textbf{call}\ p \{ \mathcal{P}, \sigma \} \rightarrow \tau$$

belongs to $\mathcal{A}'$ implies that $r \{ \sigma, \tau \}$ also belongs to $\mathcal{A}'$. This is tantamount to saying that the rule :

$$[1] \quad \frac{\textbf{call}\ p \{ \mathcal{P}, \sigma \} \rightarrow \tau}{r \{ \sigma, \tau \}}$$

is a valid production in $\mathcal{A}'$. It is easy to see that the validity of [1] implies that r is monotone.

It is interesting to study the converse of rule [1] :

$$[2] \quad \frac{r \{ \sigma, \tau \{}{\textbf{call}\ p \{ \mathcal{P}, \sigma \} \rightarrow \tau}$$

If τ is replaced by a structure, then the validity of [2] implies that procedure p terminates. One is then led to the rule :

$$[3]\quad \frac{r\,\{\,\sigma,\tau\,\}\quad \textbf{structure}\,[\tau]}{\textbf{call}\,p\,\{\,\mathcal{F},\sigma\,\}\to\tau}$$

Theorem I of paragraph 3 implies that τ in the conclusion of rule [3] is unique. Hence if [3] is valid, r is a functional relation in its second argument.

When [1] and [3] are both valid, one may assert that the procedure p and the relation r define one and the same monotone function.

When r is not functional, rule [1] may still be used to state weaker properties of p.

As an exemple, let us consider the extension, $\mathcal{A}_{\mathbf{Q}}$ of $\mathcal{A}$ containing the rule :

$$[4]\quad \frac{\begin{array}{l} x\,\{\,\sigma\,\}\to\alpha \\ y\,\{\,\tau\,\}\to\beta \\ \textbf{le}\,\{\,*\,\{\,\beta,\beta\,\},\alpha\,\}\quad \textbf{gt}\,\{\,*\,\{\,+\,\{\,\beta,1\,\},+\,\{\,\beta,1\,\}\,\},\alpha\,\}\end{array}}{\mathbf{Q}\,\{\,\sigma,\tau\,\}}$$

Q is clearly monotone. **Q** expresses the fact that the number associated to " y " is the integer square root of the number associated to " x ". If $\mathcal{F}$ is a program containing a procedure *sqrt*, then to assert the validity of :

$$[5]\quad \frac{\textbf{call}\ \mathit{sqrt}\,\{\,\mathcal{F},\sigma\,\}\to\tau}{\mathbf{Q}\,\{\,\sigma,\tau\,\}}$$

is to say that *sqrt* " extracts the square root of x and affects its value to y ". However, there is no indication in [5] about the fate of other identifiers in σ, and in fact **Q** is not functional in τ. To obtain the desired result, one must use the stronger relation **R** :

$$[6]\quad \frac{\begin{array}{l} x\,\{\,\sigma\,\}\to\alpha \\ \textbf{le}\,\{\,*\,\{\,\beta,\beta\,\},\alpha\,\}\quad \textbf{gt}\,\{\,*\,\{\,+\,\{\,\beta,1\,\},+\,\{\,\beta,1\,\}\,\},\alpha\,\} \\ y:=\beta\,\{\,\pi,\sigma\,\}\to\tau\end{array}}{\mathbf{R}\,\{\,\sigma,\tau\,\}};$$

The functionality of **R** may be derived from elementary arithmetic (β is unique) and from Theorem I. Hence the correctness of the procedure *sqrt* is expressed by the validity of the two rules

$$[7]\quad \frac{\textbf{call}\ \mathit{sqrt}\,\{\,\mathcal{F},\sigma\,\}\to\tau}{\mathbf{R}\,\{\,\sigma,\tau\,\}},$$

$$[8]\quad \frac{\mathbf{R}\,\{\,\sigma,\tau\,\}\quad \textbf{structure}\,[\tau]}{\textbf{call}\ \mathit{sqrt}\,\{\,\mathcal{F},\sigma\,\}\to\tau}.$$

It is possible to extend the method to other types of programs. One may consider, for instance, the problem of justifying a program which sorts in ascending order the integers in an array " x ". The following four rules :

$$[9]\quad \frac{\textbf{integer}\ [\alpha]}{\mathbf{s}[\sqcup\alpha, \sqcup\alpha]}$$

$$[10]\quad \frac{\{\ \alpha, \beta\ \}\quad \mathbf{s}[\gamma, \sqcup\beta\delta]}{\mathbf{s}[\gamma\sqcup\alpha,\ \sqcup\alpha\sqcup\beta\delta]}$$

$$[11]\quad \frac{\textbf{le}\ \{\ \beta, \alpha\ \}\quad \textbf{et}\ \{\ \alpha, \gamma\ \}\quad \mathbf{s}[\delta, \varepsilon\sqcup\beta\sqcup\gamma\eta]}{\mathbf{s}[\delta\sqcup\alpha,\ \gamma\sqcup\beta\quad \alpha\sqcup\gamma\eta]}$$

$$[12]\quad \frac{\textbf{le}\ \{\ \beta, \alpha\ \}\quad s[\gamma, \delta\sqcup\alpha]}{\mathbf{s}[\gamma\sqcup\alpha,\ \delta\sqcup\beta\sqcup\alpha]}$$

define a relation **s** between integer vectors which is asserted if the second argument is a sorted permutation of the first one. **s** is clearly functional in its second argument. Let σ be a data structure which contains an entry " x " and an entry " n " for the length of the initial string to be sorted in x. The result of such a sort is related to σ by the relation **S** defined by the rule :

$$[13]\quad \frac{\begin{array}{ll}\textbf{value}\ \{\ \sigma, "x"\ \} \to \alpha & n\ \{\ \sigma\ \} \to \nu \\ \textbf{index}\ \{\ \alpha, 0, \nu\ \} \to \beta & \mathbf{s}[\beta, \gamma] \\ \multicolumn{2}{c}{\textbf{modify}\ \{\ \sigma, "x", \textbf{set}\ \{\ \alpha, \gamma, 0, \nu\ \}\ \} \to \tau}\end{array}}{\mathbf{S}\ \{\ \sigma, \tau\ \}}$$

Let us now consider the following very simple sorting program :

```
𝔉 = procedure sort
    begin integer i; integer j; integer s;
        if n gt 1 then n : = n − 1;
                       call sort;
                       i : = 0;
                       while i lt n and x(i) le x(n) do i : = i + 1 endwhile;
                       s : = x(n); j : = n;
                       while j gt i do x(j) : = x(j − 1); j : = j − 1; endwhile;
                       x(i) : = s;
                       n : = n + 1
                   else endif end.
```

Let $\mathcal{A}_{\mathbf{S}}$ be the Post System defined by the rules of $\mathcal{A}$ and rules [9] to [13].

To prove the correctness of $\mathfrak{F}$, one must show that

$$[14]\quad \frac{\textbf{call}\ \textit{sort}\ \{\ \mathfrak{F}, \sigma\ \} \to \tau}{\mathbf{S}\ \{\ \sigma, \tau\ \}}$$

and

$$[15]\quad \frac{\mathbf{S}\ \{\ \sigma, \tau\ \}\quad \textbf{structure}\ [\tau]}{\textbf{call}\ \textit{sort}\ \{\ \mathfrak{F}, \sigma\ \} \rightarrow \tau}$$

are valid rules in $\mathcal{A}_{\mathbf{S}}$.

In $\mathcal{A}_{\mathbf{S}}$, the only way to prove a word containing the letter S is by an instance of rule [13]. Hence a proof of $\mathbf{S}\ \{\ \sigma, \tau\ \}$ is the combination of one proof of each of the antecedents of [13] followed by an application of [13]. In such a proof, let N be the integer to be substituted for the syntactic variable ν. Suppose first that N is 1. From the definition of the **index** function (see rule 3.3 [8]) one may conclude that the length of β is one. In this case, rule [9] shows that γ is identical to β, and hence that σ is identical to τ. But from the proof of

$$n\ \{\ \sigma\ \} \rightarrow 1$$

it is possible to deduce a pròof for

$$n\ \textbf{gt}\ 1\ \{\ \sigma\ \} \rightarrow \textbf{false}.$$

In this case, then, the body of *sort* is equivalent to an empty statement; this fact constitutes a proof of

$$\textbf{call}\ \textit{sort}\ \{\ \mathfrak{F}, \sigma\ \} \rightarrow \sigma$$

Rule [15] is therefore valid.

Suppose now that the validity of [15] has been proved for an integer N. Let Σ be a data structure for which

$$[16]\quad n\ \{\ \Sigma\ \} \rightarrow N + 1.$$

Let $\mathcal{B}_1$ be an abbreviation for the body of *sort*. To compute the result of an application of *sort* to Σ, one must first evaluate $\mathcal{B}_1\ \{\ \mathfrak{F}, \Sigma \not\equiv\ \}$ (see rule 3.346 [6]). $\mathcal{B}_1$ begins by declarations for i, j and s. After execution of these statements, one obtain :

$$\Sigma_1 \equiv \Sigma \not\equiv i\ \textbf{integer}\ (\sqcup\ \infty)\ j\ \textbf{integer}\ (\sqcup\ \infty)\ s\ \textbf{integer}\ (\sqcup\ \infty).$$

None of these identifiers are identical to n or x. Hence from [16] one may deduce

$$n\ \textbf{gt}\ 1\ \{\ \Sigma_1\ \} \rightarrow \textbf{true}\ ;$$

this implies that the **if** statement is equivalent to its **then** part. Let

$$\Sigma_2 \equiv n :=\ n - 1\ \{\ \mathfrak{F}, \Sigma_1\ \};$$

it is clear that

$$n\ \{\ \Sigma_2\ \} \rightarrow N.$$

By hypothesis, there exists a proof of $\mathbf{S}\ \{\ \Sigma, T\ \}$ for a particular T. This proof contains a subproof of $\mathbf{s}\ \{\ B, \Gamma\ \}$ for an integer vector B defined by

$$\textbf{index}\ \{\ \textbf{value}\ \{\ \Sigma, ''x''\ \}, O, N + 1\ \} \rightarrow B$$

and a particular Γ. It is easy to see that by omitting the last step in this proof, one gets a proof of

$$\mathbf{s}\ \{\ B', \Gamma'\ \}$$

for a B' defined by :

$$\textbf{index}\ \{\ \textbf{value}\ \{\ \Sigma, ''x''\ \}, O, N\ \} \rightarrow B'$$

and another well defined vector Γ'.

The structure Σ_3 defined by

$$\textbf{modify}\ \{\ \Sigma_2, ''x'', \textbf{set}\ \{\ \textbf{value}\ \{\ \Sigma_2, ''x''\ \}, \Gamma', O, N\ \}\} \rightarrow \Sigma_3$$

is such that

$$\mathbf{S}\ \{\ \Sigma_2, \Sigma_3\ \}$$

and hence, by the recursion hypothesis :

$$\textbf{call}\ \textit{sort}\ \{\ \mathfrak{F}, \Sigma_2\ \} \rightarrow \Sigma_3.$$

Let $\mathfrak{B}_2$ be the remaining instructions in $\mathfrak{B}_1$. Let Σ_4 be defined by :

$$\mathfrak{B}_2\ \{\ \mathfrak{F}, \Sigma_3\ \} \rightarrow \Sigma_4.$$

There will be three distincts case according to whether the last rule in the proof of $\mathbf{s}\ \{\ B, \Gamma\ \}$ is [10], [11] or [12]. In the first case, the first **while** of $\mathbf{B}_2$ will fail on the first pass and the value associated to i will be O. In the second case, this **while** fail because for some i $x(i)$ will be greater than $x(n)$. In the third case, the **while** will terminate when i reaches the value N. In all three cases, the second **while** will insert $x(n)$ in its proper place, the net result beeing that

$$\textbf{index}\ \{\ \textbf{value}\ \{\ \Sigma_1, ''x''\ \}, O, N + 1\ \} \rightarrow \Gamma,$$

while the last statement of $\mathfrak{B}_2$ has restored n to its original value. It is clear then that Σ_4 is built by concatenating T, the sign $\not\equiv$ and the entries for i, j and s. As these last elements are eliminated by an application of 3.486 [6], the net results is a proof of

$$\textbf{call}\ \textit{sort}\ \{\ \mathfrak{F}, \Sigma\ \} \rightarrow T.$$

Hence [15] is also valid in this case.

The above discussion is not formal enough to really constitute a proof for the sort procedure. The building of a formal system in which to express such proofs will be the subject of further research.

5. DISCUSSION

5.1. This work is intended as a first step in the definition of methods for proving the correctness of a program. The need for such methods is clear if one considers the present trend toward faster computers executing more complicated software.

The basic problem here is the elimination of human programming errors. As in all such cases, there are two possible approaches. One is to attempt to streamline the task of the programmer; this approach has lead to the use of higher and higher level languages. The other way attempts to introduce redundancy in the definition of a program; this may be done by giving an explicit algorithm and a sufficient set of properties. Any discrepancy between these two definitions indicates an error. Note that the absence of a discrepancy is not an absolute proof of correctness : both definition may contain compensating errors; however this event is of low probability; in fact a method to lower this probability is to use widely different definitions, perhaps constructed by different peoples.

If the redundancy check is to be any indication of correctness, it must be as error free as possible. This implies that any method used in proving properties of programs must be automatic or semi-automatic and hence that the proof techniques must be completely formal.

This research program clearly has three main parts :

1) to prove anything about a program, one must known its meaning : hence the semantic of its programming language must be defined in a completely formal way;

2) in the same fashion, one must have a language for stating properties of programs and a completely formal semantic for it;

3) the last part is the construction of a deductive system for proving or disproving properties of programs.

This paper is concerned only with parts (1) and (2) and attempts to show that Post systems may be used as underlying formal systems for this work. One should note that there exists quite a few types of constructive formal systems and that all universal ones (the general recursive functions, Markov algorithms, Turing machines, combinatory arithmetic, Post languages) were shown to be mutually equivalent. Hence the choice of such a system is necessarily based on pragmatic reasons.

Post grammars seem to be a good compromise between inherent simplicity and expressive power. In this connexion, one may note that the number of semantical production in chapter 3 is of the same order of magnitude as the number of syntactical ones.

On the other side, Post grammars have no internal constraints on the construction and manipulation of formal entities; this is certainly an advantage and allows one to test quite freely any number of logical systems.

It is clear that the price of this versatility is a decreased efficiency and that operational systems of the future will be built on fixed *ad hoc* formal systems.

5.2. Current research on program correctness follows three principal directions :

1) the method of Floyd [5], as developed for instance by Hoare and his co-workers (See for instance Hoare [6] and specially Hoare and Wirth [7]);

2) the use of the system LCF (Logic for computable Functions). See for instance Milner [11];

3) operational semantic, as exemplified by the Vienna method (Lucas and Walk [10]).

The method of Floyd answers in principle to all three objectives of paragraph 5.1. The basic idea of Floyd was to use the formal variables of a first order calculus as a representation of the « variables » of a programming languages (i.e. of names of cells in the computer storage). While this method works well for individual cells, it is clear (see for instance Burstall [1]) that it cannot cope in a natural way with any kind of address computation (indexing, indirect addressing, pointers etc.). Attempts to solve this problem in Hoare and Wirth [7] are not completely formal; the resulting system is no longer a first order calculus in the strict sense.

The LCF language is primarily a system for naming general recursive functions, that is to say functions constructed from primitives by composition, abstraction, conditional evaluation and resolution of recursion equations. LCF is built on a system of type closely related to the theory of functionality of Curry ([3] chapter 11). (In the present case, the interest of this type system is doubtful; among other problems, it is difficult to correlate the types of LCF with the ordinary types of a conventional programing languages; in fact, many useful functions (like **value** in paragraph 3.32) have no definite type in this case).

LCF contains also axioms and rules of inference allowing one to prove the equivalence of two functions. To prove a property of a program, one must :

– associate a LCF function to it;

– associate a LCF relation to the property to be proved;

– prove the compatibility of these two objects (i.e. if f is the function and r the relation, prove that the relation $\{ \lambda x \,.\, r(x, f(x)) \}$ is equivalent to the constant relation **true**).

The only difficulty in the use of LCF is the translation between concrete and abstract forms of programs and properties, which must be done outside of the formal system.

The Vienna method deals only with the first part of the program outlined in paragraph 5.1. The method used is similar to the present one : the meaning of a program is defined by an interpreter acting on abstracts representation of the program and its data. As in the case of LCF, one needs an *ad hoc* translator to build the abstract representation of a concrete program. The system used to describe this interpreter is not completely formal. However, it seems probable that such a formalization could be easily carried out in LCF or any like medium. A difference with the present approach is that the meaning of a program is not defined by its ultimate result, but by a state change function; the result, if it exists, is the first fixed point of this function.

There is no indication that an attempt was made to extend the Vienna Method to prove properties of programs. One many remarks in this connexion that the choice of the state change function as the main semantical tool would lead to the use of invariance properties.

5.3. Directions for future research

In the present context, the next step towards the program outlined in 5.1 is the building of a formal deductive system for program properties. One may either look for a general solution (to prove the validity of a rule in an arbitrary Post system) or capitalize on the special properties of grammar $\mathcal{A}$. One may note that while the detailed structure of $\mathcal{A}$ is quite arbitrary, any system able to describe algorithms must have some kind of definiteness property as expressed by Theorems I to III.

If such a formal system is built, its proofs will very probably be very long and tedious; any practical use will depend on the availability of mechanical aids. Here again one may consider general tools (theorem provers in an arbitrary Post system) or special purpose theorem provers.

Lastly, one may ask if its is possible to build in a more or less automatic way an algorithm having a given set of properties; this is a very difficult problem if one exclude trivial exhaustive methods. It is however known that theoretical solution exists for particular cases (see for instance Waldinger and Lee [15]).

I would like to thank J. Arsac for many stimulating discussions and specially L. Nolin for sowing the germ of an idea which led to the consistency proof of paragraph 3.5.

APPENDIX A

Some additional syntactical rules.

[1] $\dfrac{}{\textbf{constructor [length]}}$

[2] $\dfrac{}{\textbf{constructor [value]}}$

[3] $\dfrac{\textbf{reference } [\rho]}{\textbf{construct } [\rho]}$

[4] $\dfrac{}{\textbf{constructor [name]}}$

[5] $\dfrac{\textbf{program } [\pi]}{\textbf{construct } [\pi]}$

[6] $\dfrac{\textbf{expression } [\varepsilon]}{\textbf{constructor } [\varepsilon]}$

[7] $\dfrac{\textbf{body } [\beta]}{\textbf{constructor } [\beta]}$

[8] $\dfrac{\textbf{structure } [\sigma]}{\textbf{construct } [\sigma]}$

[9] $\dfrac{\textbf{type } [\tau]\ \tau[\chi]}{\textbf{construct } [\chi]}$

[10] $\dfrac{\mu - \nu \textbf{ operator } [\omega]}{\textbf{constructor } [\omega]}$

[11] $\dfrac{}{\textbf{constructor [index]}}$

[12] $\dfrac{}{\textbf{constructor [set]}}$

[13] $\dfrac{}{\textbf{constructor [modify]}}$

APPENDIX B

Operators.

" – ' **operator** [*]

" – ' **operator** [/]

" – ' **operator** [%] (remainder)

' – " **operator** [–]

" – " **operator** [–]

" – " **operator** [+]

' – "" **operator** [↓] (integer to decimal string conversion)

" – "" **operator** [|] (string concatenation)

' – """ **operator** [↑] (decimal string to integer conversion)

" – """ **operator** [gt]

" – """ **operator** [ge]

" – """ **operator** [eq]
" – """ **operator** [ne]
} (these two operaton au defined both on int egers and strings)

" – """ **operator** [le]

" – """ **operator** [lt]

' – """" **operator** [not]

' – """" **operator** [and]

' – """""" **operator** [or]

REFERENCES

[1] R. M. BURSTALL, Some techniques for proving correctness of programs which alter data structures, *Machine Intelligence*, **7** (1972), 23-50.

[2] H. B. CURRY, Foundations of Mathematical Logic, Mac Graw Hill, New York, 1963.

[3] H. B. CURRY and R. FEYS, Combinatory Logic, vol. I. North Holland, Amsterdam, 1958.

[4] P. FEAUTRIER, Introduction aux langages de Post en tant qu'outil de formalisation. Publication de l'Institut de Programmation de l'Université Paris VI, n° 74-7, Paris, 1974.

[5] R. FLOYD, Assigning Meanings to Programs, *Proc. Amer. Math Soc. Symposia on Applied Math*, **19** (1966), 19-32.

[6] C. A. R. HOARE, An Axiomatic Basis for Computer Programming, *Comm. Assoc. Comp. Mach*, **12** (1969), 576-583.

[7] C. A. R. HOARE and N. WIRTH, An Axiomatic Definition of the Programming language Pascal, *Acta Informatica*, **2** (1973), 335-355.

[8] D. E. KNUTH, Semantics of context free languages, *Math. Syst. Th.*, **2** (1968), 127-145.

[9] H. F. LEDGARD, Production systems : or can we do better than BNF, *Comm. Assoc. Comp. Mach*, **17** (1974), 94-102.

[10] P. LUCAS and K. WALK, On the formal Description of PL/I, *Ann. Rev. on Automatic Programming*, **6** (1969), 105-182.

[11] R. MILNER, Implementation and Applications of Scott's LCF. *Proc. Conf. on Proving Assertions about Programs*, New Mexico State U, 1972, 1-6.

[12] L. NOLIN, Formalisation des notions de machine et de programme, Paris, Gauthier Villard, 1968.

[13] E. POST, Formal Reduction of the general combinatorial decision problem, *Amer. J. Math*, **65** (1943), 197-215.

[14] P. C. ROSENBLOOM, The Elements of Mathematical Logic, Dover, New York, 1950.

[15] R. J. WALDINGER and R. C. T. LEE, PROW, a step toward automatic program writing, *Proc. Int. J. Conf. on Artificial Intelligence*, N. Y., 1969, 241-252.