ALGEBRAIC ASPECTS OF ALGOL 68[†]

Suad Alagić

Computer and Information Science
University of Massachusetts
Amherst, Mass. 01002

COINS Technical Report 73B-5
November, 1973

ABSTRACT

The structure of Algol 68 is investigated from the algebraic viewpoint. The rules for composing modes are axiomatized using existing algebraic machinery. Actions corresponding to standard programming phrases and the storage model are constructed from these rules of data structuring only. The category of sets, the category of pointed sets, and the category of complete lattices and continuous functions are studied as candidates for the ideal algebraic model. It is shown what the advantages and the drawbacks are of the lattice-theoretic approach over the classical partial function approach for an algebraic theory of programming languages with the structure of Algol 68.

[†]The research reported in this paper was supported by the National Science Foundation under Grant No. GJ 35759. The author gratefully acknowledges the help which he received from Dr. Michael Arbib, Dr. Robert Taylor and Dr. Ernest Manes, and the comments given by Mr. Stephen Hegner.

Introduction

The objective of this paper is to show that Algol 68 has the essential features necessary for the development of an elegant, intelligible, algebraic theory of programming. Most fundamental concepts of the language--in particular the rules for composing modes--can be carefully axiomatized using existing algebraic machinery. We show to what extent the lattice-theoretic approach as well as the classical partial function approach are relevant to a language with the structure of Algol 68 which, in spite of its apparent complexity, turns out to be very systematic and mathematically interesting. It is our hope that the detailed presentation of this last statement given in the paper is a modest contribution to resolving some of the controversy of opinions about Algol 68.†

1. Composition of Modes

We call a well-formed piece of programming text a *phrase*. Phrases are *external* objects and can be elaborated upon by some assumed abstract interpreter. The result of *elaboration* is an *action* (of the interpreter). Specifying to what actions syntactically valid phrases give rise when elaborated upon, is one possible way of *semantic definition* of the language.

Values are internal objects upon which actions operate. A set of values which share some common properties is characterized by a *mode*.

---

†The readers not familiar with the language are referred to [3]. We make no claims that in our discussion of Algol 68 we stick strictly to the language definition in [20].

We consider four primitive modes of Algol 68, denoted externally as real, int, bool and char.† Out of these one can construct recursively new modes according to the five rules to which should correspond definite algebraic operations on sets of values. So for example if amode and bmode are modes then [] amode is a new mode of multiple values (arrays) of mode amode. struct (amode, *first*, bmode *second*) is a mode of structured values or records with the two fields of respective modes amode and bmode. union (amode, bmode) characterizes values which can be either of amode or bmode but not both. proc (amode) bmode is a mode of routines (which are therefore values, too) with one formal parameter of amode and delivering a result of mode bmode. ref amode stands for the set of names or references to the values of amode. Examples of modes constructed in this manner are ref [] real, proc (ref int, real) real, struct (char *first*, [] union (char, int) *second*) etc.

The above rules can be captured in algebraic axioms in such a way that the new modes (sets of values) constructed out of the already given ones can be defined in terms of actions of the abstract interpreter only. In other words, the rules of data structuring are defined in terms of what can be done with so structured data. We argue that, in a well founded theory of programming, actions obtained by the elaboration of standard programming phrases, for example conditional actions, repetitive actions, etc. should be constructed from the axioms rather than by ad hoc definitions. To accomplish this goal, we might have to impose a

---

†Algol 68 features also the fifth primitive mode: format. Values of mode format are used for the control of input-output.

certain structure on the sets of values, for example the structure of a complete lattice. These sets with (possibly) some structure are called *objects*.

A convenient framework for talking about objects and actions is a *category*. Its definition reflects our emphasis on sets of values (objects) and actions, rather than on syntax of phrases.

(1.1) A category consists of a collection of objects A, B, C, ..., together with two functions, as follows:

(i) A function assigning to each pair (A,B) of objects a set act (A,B). An element $f \in act(A,B)$ in this set is called an action[†] $f : A \to B$ with domain A and codomain B.

(ii) A function assigning to each triple (A,B,C) of objects a function

$$act(B,C) \times act(A,B) \to act(A,C)$$

For actions $g : B \to C$ and $f : A \to B$ this function is written $(g,f) \to g \cdot f$ and $g \cdot f$ is called the (serial) *composition* of f with g. Moreover, the following two axioms must hold:

(iii) The composition of actions is *associative*,[††] i.e., if $h : C \to D$, $g : B \to C$ and $f : A \to B$ are actions with indicated domains and codomains, then

$$h \cdot (g \cdot f) = (h \cdot g) \cdot f$$

(iv) For each object A there exists an *identity* action $1_A : A \to A$ such that for $f : A \to B$, $g : C \to A$, $f \cdot 1_A = f$ and $1_A \cdot g = g$.

[†] In the normal usage of category theorists what we termed an action is called a morphism.

[††] In the real world this is, of course, only approximately true, but it seems to be an assumption adopted in other mathematical approaches to programming languages (cf. [14], p. 531).

If actions f and g are obtained by the elaboration of phrases e_1 and e_2 respectively then their composition $g \cdot f$ is denoted externally (i.e., in the language) as $e_1; e_2$. The identity action is denoted with skip.

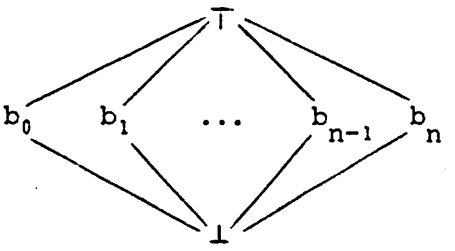
For our discussion the three main examples of categories are going to be Set, $\underline{Set}_*$ and Clatt. Set has objects-sets and actions-total functions. $\underline{Set}_*$ has objects-pointed sets. By a *pointed set* is meant a nonempty set with a selected element written * and called a *base point*. An action $f : A \to B$ in $\underline{Set}_*$ is a function on the pointed set A to the pointed set B which carries the base point of A to the base point of B. $\underline{Set}_*$ is of extreme importance; it is isomorphic to the category Pfn of sets and partial functions.† To see this, note that a partial function $f : A \to B$ can be represented as a total, base point preserving function $f^* : A \cup \{*_A\} \to B \cup \{*_B\}$. For $a \in A$ $f^*(a)$ is defined to be equal to $f(a)$ whenever $f(a)$ is defined and $f^*(a) = *_B$ if not. Moreover, we set $f^*(*_A) = *_B$. Conversely, given an action $h^* : C \to D$ in $\underline{Set}_*$ we can define a partial function $h : C \setminus \{*_C\} \to D \setminus \{*_D\}$ in such a way that these two passages are inverse to each other.

Denote the base point in $B \cup \{*_B\}$ by $\perp$. We can impose a partial order on the set $B \cup \{\perp\}$ making $\perp$ the lower unit (totally undefined element, less than any other element). Furthermore, for the sake of mathematical symmetry and some technical requirements to be explained later, we also adjoin an upper unit $\top$ and get a structure which is a simple example of what is known as a complete lattice:††

---

†This is important to bear in mind since we are going to use $\underline{Set}_*$ rather than Pfn for aesthetic reasons.

††The figure represents the partial order under which the elements of B are incomparable.



The formal definitions follow:

(1.2) (i) If $(P,\leq)$ is a partially ordered set and A is any subset, then the element c is a *least upper bound* (supremum, join) of A, denoted $c = l.u.b.A = supA$ if 1) $a \leq c \quad \forall a\in A$, 2) $(a\leq x, \forall a\in A)$ implies $c \leq x$.

(ii) A *complete lattice* is a partially ordered set $(L,\leq)$ such that every subset of L has a least upper bound. In particular, for the empty subset $\phi \subset L$ denote $\bot = sup\phi$, the lower unit of L and $\top = supL$, the upper unit of L.[†]

(iii) A subset D of a complete lattice $(L,\leq)$ is *directed* if any finite subset of D has a least upper bound in D. Since this applies to ϕ it follows that D, continuing ϕ's sup, must be nonempty.

(iv) Let $(L,\leq)$, $(L',\leq)$ be complete lattices. A function $f : L \to L'$ is *continuous* if whenever $D \subset L$ is directed, we have

$$f(supD) = sup\{f(x) : x\in D\}$$

Continuous functions are monotonic, i.e., for $x \leq y$ we have $f(x) \leq f(y)$.

Denote with Clatt the category which has complete lattices as objects and continuous functions as actions. This category was introduced by Scott with the goal to capture in a more satisfactory way the classical partial function theory of computation.

[†]This definition of a complete lattice implies that greatest lower bounds (meets, infs) exist. For $D \subseteq L$ $infD = sup\{x\in L : x\leq y \text{ for all } y\in D\}$ (cf. [1], p. 126 for the proof).

(1.1) states the basic properties a category should satisfy. Our goal is to formulate, according to Algol 68, new axioms and investigate which categories (in particular Set, Set_*, Clatt) satisfy them. We first look carefully at structured values of Algol 68.

Given modes ml, ..., mn and identifiers *s1*, ..., *sn* the phrase struct (ml *s1*, ..., mn *sn*) characterizes the object of *structured values* whose components are called *fields*. The i-th field has mode mi and is selected by the identifier *s1*. Structured values are also called records, for example struct ([] char *name*, bool *sex*, int *age*) characterizes an object of personal records and struct (real *re*, real *im*) denotes an object whose elements are complex numbers. This external, syntactic description in terms of phrases is formalized in the following semantic axiom involving actions.

(1.3) Given objects M_1 and M_2, there exists an object $M_1 \times M_2$ together with two actions $p_i : M_1 \times M_2 \to M_i$ (i=1,2) called projections, p_i corresponding to *si*. If, externally, M_1 and M_2 have modes ml and m2 respectively, then we say that $M_1 \times M_2$ has mode struct (ml *s1*, m2 *s2*). $M_1 \times M_2$ is required to satisfy the following property:

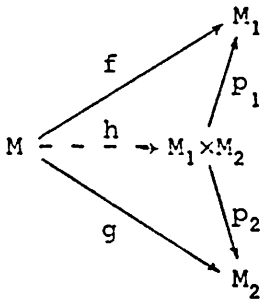
Given any other object M and a pair of actions $f : M \to M_1$, $g : M \to M_2$, there exists a *unique* action $h : M \to M_1 \times M_2$ such that

$$p_1 \cdot h = f \quad \text{and} \quad p_2 \cdot h = g$$

The above statement is represented by the diagram which we say commutes.†

---

†This construction is known to category theorists as a product. The two triangles of the following diagram represent graphically the fact that the composite actions $p_1 \cdot h$ and $p_2 \cdot h$ are equal to f and g respectively.



h will be denoted as (f,g).

The above axiom generalizes in an obvious way to a finite family $\{M_i\}$ of objects. If we require the axiom to hold even if we take an empty family of objects, then we get:

(1.4) There exists a terminal object I with the property that for any other object M there exists a unique action $t_M : M \to I$.

I does not have a mode.†

It is easy to show that (1.3) and (1.4) determine $M_1 \times M_2$ and I uniquely up to isomorphism. Here by an isomorphism of A and B we mean an action $f : A \to B$ such that there exists an action $g : B \to A$ satisfying $g \cdot f = 1_A$, $f \cdot g = 1_B$.

Consider now whether (1.3) and (1.4) are satisfied in Set, Set_* and Clatt. In Set $M_1 \times M_2$ is just the ordinary cartesian product of sets and I is just a one element set. But we know that real programs in general compute not total, but partial functions, because of possible non-terminating loops. Therefore Set is not really a good model, and for that reason consider Set_*. Given two pointed sets M_1 and M_2, $M_1 \times M_2$

---

†In certain categories I has only one value which is interpreted as the unspecified value of the mode required by the context.

is again the cartesian product of sets. The base point of $M_1 \times M_2$ is $(*_{M_1}, *_{M_2})$ where $*_{M_1}$ and $*_{M_2}$ are base points of M_1 and M_2 respectively. The terminal object is the set consisting of the base point only.† For two complete lattices M_1 and M_2, define the partial order on the cartesian product $M_1 \times M_2$ of sets as

$$(x_1, y_1) \leq (x_2, y_2) \text{ if and only if } (x_1 \leq x_2 \text{ and } y_1 \leq y_2)$$

For $D \subseteq M_1 \times M_2$ let D_1 and D_2 be the respective images of D under projections $p_1 : M_1 \times M_2 \to M_1$ and $p_2 : M_1 \times M_2 \to M_2$. Then define $\sup D = (\sup D_1, \sup D_2)$. With these definitions $M_1 \times M_2$ becomes a complete lattice. To prove that (1.3) holds one can establish that projections are continuous and that the unique function h in (1.3) is indeed continuous (cf. [19]). The terminal object I in Clatt is a one element lattice in which $\bot = \top$. It is easy to see that the unique function $M \to I$, where M is any complete lattice, is indeed continuous.

The next crucial construction is uniting. Given modes m1, ..., mn one can form a new mode union (m1, ..., mn). This new mode characterizes an object consisting of values which are either of mode m1 or of mode m2 . . . or of mode mn but no value can be both of modes mi and mj with $i \neq j$. An example of mode declaration involving unions is

mode s-expr = union (atom, struct (ref s-expr *first*, ref s-expr *second*)††

The above mode declaration defines McCarthy's *s-expressions*, a data

---

†Observe that in Pfn if $M_1 \times M_2$ and I are defined as the set of ordered pairs and the one element set respectively, both (1.3) and (1.4) fail.

††The problem of null reference is dealt with in Algol 68 using nil which is of mode ref amode for any amode. It yields a reference which does not refer to any value.

structure which is the basis for Lisp 1.5. It says that an s-expression is either an atom or it is a pair of (references to) s-expressions. This example shows that uniting is implicit in other programming languages with systematic structure (like Lisp 1.5). It is Algol 68 that makes this an explicit rule of data structuring.†

Another example illustrating the importance of this concept of Algol 68 follows.†† It is a procedure which delivers the exact integer n! as long as this value remains less than the maximal representable integer (within a particular environment), otherwise a procedure *faclarge* is called to give a reasonable approximation, which is in that case of mode real.

```
mode intreal = union (int, real);
proc factorial = (int n) intreal:
                    if n > nmaxfac
                    then faclarge (n)
                    else int f := 1;
                         for i from 2 to n do f := f×i; f
                    fi;
```

One can actually find out whether the delivered value is exact or not by testing its mode.†††

Uniting is captured in the following algebraic axiom:

(1.5) Given objects M_1 and M_2, there exists an object M_1+M_2 together with a pair of actions $i_k : M_k \to M_1+M_2$ (k=1,2). If, externally, M_1 and

---

†We show later in the paper that this is very important from the mathematical viewpoint.

††The example is from [12].

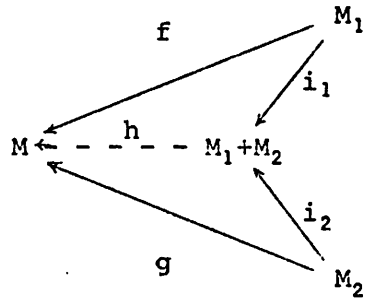
†††In the above example as well as in some others that follow we use phrases corresponding to actions not yet defined. By the end of the paper all of them will be discussed and formally established, at least in some limited form.

M_2 have modes m1 and m2 respectively, then we say that M_1+M_2 has mode union (m1, m2). M_1+M_2 is required to satisfy the following property:

Given any other object M and a pair of actions $f : M_1 \to M$ and $g : M_2 \to M$ there exists a unique action $h : M_1+M_2 \to M$ such that

$$h \cdot i_1 = f \quad \text{and} \quad h \cdot i_2 = g$$

i.e., the following diagram is commutative:†



Actions i_1 and i_2 are called injections and the above statement determines M_1+M_2 unique up to isomorphism. It also generalizes easily to an arbitrary finite family of objects. We denote h as $\binom{f}{g}$.

Note that the commutative diagram defining M_1+M_2 in (1.3) looks like the diagram defining M_1+M_2 in (1.5) save that the arrows are reversed. We thus say that the constructions are *dual*, conforming to the usage of category theorists.

However, (1.5) is not strong enough and we strengthen it requiring that it holds even if we take a countable family $\{M_i\}$ of objects. The following example illustrates why this is important.

mode string = [1 : 0 flex] char

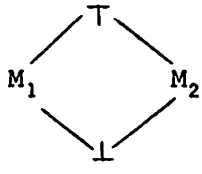
---

†This construction is known to category theorists as a coproduct.

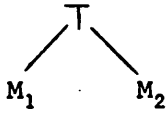
The above mode declaration says that string is of mode row of characters, their number being flexible and initially set to 0 (so that initially we have the empty string). We can say that string is a countable union of modes *[1; k]* char, *k=0,1,2,...* and that's why we need the stronger axiom than (1.5).

In Set (1.5) is satisfied with M_1+M_2 being a *disjoint union* of M_1 and M_2. In Set_* M_1+M_2 is a disjoint union of M_1 and M_2 except that their base points are identified. The reader can easily check this. However, in Clatt only a weaker form of (1.5) is satisfied.

The best we can do in constructing M_1+M_2 is to choose a disjoint union of M_1 and M_2 with new lower and upper units added and the partial order defined pictorically as:



Obvious injections of M_1 and M_2 into M_1+M_2 are clearly continuous. For given two continuous functions $f : M_1 \to M$ and $g : M_2 \to M$ one can in general define more than one continuous function $h : M_1+M_2 \to M$ such that the diagram in (1.5) commutes. Among these it is natural to choose the least one as $\binom{f}{g}$. This is possible since the set of all continuous functions with domain M_1+M_2 and codomain M is a complete lattice itself, as will be explained later. The function h that we want is constructed as follows. Define $h(x) = f(x)$ if $x \in M_1$ and $h(x) = g(x)$ if $x \in M_2$. Since



is directed and its sup is $\top$ for h to be continuous we have to set $h(\top) = \sup\{h(x) : x \in M_1 \cup M_2\}$.† Finally, $h(\bot)$ is not uniquely determined, since ϕ is not directed and $\bot = \sup\phi$. The least h we discussed before is obtained by setting $h(\bot) = \bot$. Observe that this definition gives a continuous h by f and g continuous.

The construction of M_1+M_2 in Clatt exhibited above is a weaker form of (1.5) (called weak coproduct) and because of that there will be a number of unpleasant departures of Clatt from the ideal category for our purposes. While (1.5) implies that uniting is associative up to isomorphism (this can be proved and is a well known fact) for + as constructed in Clatt that is not true any more. And Algol 68 postulates the associativity of union.

(1.3) and (1.5) also establish two cases of *collateral* (i.e., parallel) *composition* of actions in Algol 68 (cf. [3], p. 705). If the actions f and g in (1.3) and (1.5) are obtained by the elaboration of the phrases e_1 and e_2 respectively, then their collateral composition in both cases will be denoted externally by the phrase (e_1, e_2). Internally, i.e., within the category, we denote them (f,g) and $\binom{f}{g}$ respectively to emphasize the duality of the two concepts.

Let R, N, C and L be objects corresponding to respective modes real, int, char and bool. With (1.4) and (1.5) two of these objects can be constructed from (1.4) applying the axiom (1.5). To appreciate this point, observe that values, i.e., elements of objects can often be represented as actions of a particular sort. Indeed, for Set, an action $I \to M$ corresponds to a unique element of M, the image of the only element of I.

---

† $M_1 \cup M_2$ denotes the union of the sets M_1 and M_2.

So elements of sets can be viewed as actions $I \to M$. For Clatt this will work too. Suppose that this is true in general, i.e., that there are enough actions $I \to M$ in the category for representing elements of M. Unfortunately, this fails in $\underline{Set}_*$ because I consists of the base point only. Since functions in $\underline{Set}_*$ are base point preserving, there is only one function $I \to M$ for any object M of $\underline{Set}_*$. On the other hand, our assumption constructs L as

$$L \cong I + I \tag{1.6}$$

Where $\cong$ denotes isomorphism. The two injections $I \overset{1}{\underset{0}{\rightrightarrows}} L$ are externally denoted as true and false respectively. In $\underline{Set}_*$ these two injections are the same (i.e., true = false). Even so we can still construct L in $\underline{Set}_*$ as $L \cong 1 + 1$ where 1 is a two element set, one of them being the base point. L now has three elements. Two of them correspond to true and false and the third is the base point. To represent elements as actions in $\underline{Set}_*$ we have to use 1 instead of I. Furthermore, we construct N as

$$N \cong I + I + I + \cdots \tag{1.7}$$

where on the right hand side we have a countable union (coproduct) of copies of I. In $\underline{Set}_*$ we would of course take 1 instead of I.

Returning to our motivating discussion of Algol 68 we recall that if $\underline{m}$ is a mode, then so are $[]\underline{m}$, $[,]\underline{m}$, $[,,]\underline{m}$, ..., characterizing sets of *multiple values*, i.e., arrays of values of mode $\underline{m}$ with 1, 2, 3, ... etc. dimensions. In Set, an array of mode $[1{:}n]\underline{m}$ is internally just a function $N' \to M$, where $N' = \{1,\ldots,n\}$ and M is a set of values of mode $\underline{m}$. Similarly, an array $[2{:}4, 3{:}7]\underline{m}$ is internally a function $N'' \times N''' \to M$ where $N'' = \{2,3,4\}$ and $N''' = \{3,4,5,6,7\}$, since it specifies, for every

pair $(a,b) \in N''\times N'''$, a value $m \in M$ of mode m. So in Algol 68 certain sets of functions are characterized by a mode. More generally, we require for two given objects M' and M" that the set of all actions act(M',M") is an object of the category and is characterized by a mode. For example, in Clatt this would mean that the set of all continuous functions with domain M' and codomain M" is again a complete lattice, which is known to be true and will be discussed later. This axiom is the first step in establishing the fact that routines are values, as they are in Algol 68.

Given modes m1, ..., mn, m the phrase proc (m1, ..., mn) m denotes a new mode. The values of this new mode are routines with n arguments of respective modes m1, ..., mn and returning a value of mode m. A *routine* is an internal representation of the phrase of the language, at the head of which appears a list of formal parameters. An example of a denotation of a routine of mode proc (real, real) real is

(real x, real y) real : (x+y)/2 - sqrt(x×y)

A routine is internally, i.e., within the category, a composition of actions and therefore an action itself. That's why we want to postulate that the set act $(M_1\times\cdots\times M_n, M)$ is an object of the category, characterized by the mode proc (m1, m2, ..., mn) m.† This will suffice for routines with no side-effects, only those can be thought of simply as functions (or actions). Generally, routines are more than just actions defined as above, but that will be discussed later after introducing the storage model of Algol 68 and procedure calls.

---

†Observe that the modes proc (struct (m1 *s1*, m2 *s2*)) m and proc (m1, m2) m would unfortunately not be distinguished internally under our *interpretation*.

The kind of closure discussed above is a very powerful postulate. In particular, it allows us to achieve the effect of what is usually termed a call by name. Denote act (M,N) by N^M for simplicity and consider the following declaration:

```
proc seriessum = (int k, proc (int) real term) real:
        begin real sum : = 0;
              for j to k do sum : = sum + term (j);
              sum
        end;
```

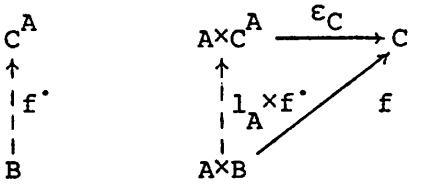
The identifier *seriessum* is made to possess a value of mode proc (int, proc (int) real) real, which is a routine summing the terms of some series from 1 to k. When it is called, two parameters are supplied, one being integer and the other one another routine, a value of mode proc (int) real. During a call of the declared procedure, the routine possessed by *term* is called once each time round the loop.† All this is consistent with our algebra if act (N × act (N,R), R) = $R^{N \times R^N}$ is an object of the category having the associated mode proc (int, proc (int) real) real.

Not only do we want to recognize routines as values, but we also want to be able to call them. Mathematically, this amounts to function evaluation modulo some other effects. The following *exponentiation axiom* takes into account all of this and more. It requires that for any objects A and B, act (A,B) is an object of the theory and that the evaluation map has certain nice properties which will be used later to construct conditional actions in certain categories.

---

†This example is taken from [12].

(1.8) Given any two objects A and C, act (A,C) = C^A is again an object of the category. There exists an action $\varepsilon_C : A \times C^A \to C$ with the property that for any object B and an action $f : A \times B \to C$ there exists a unique action $f^\cdot : B \to C^A$ such that the following diagram commutes:



(1.9)

ε_C is called the *evaluation action.*†

To see what the above axiom says consider Set. For A and C sets, C^A is just the set of all functions with domain A and codomain C. Given a function $f : A \times B \to C$ and $(a,b) \in A \times B$, we can compute $c = f(a,b)$. To this function f corresponds a unique function $f^\cdot : B \to C^A$ such that $f^\cdot(b)(a) = c$. Observe that $f^\cdot(b) : A \to C$ and that $f^\cdot(b)(a)$ makes sense.

(1.8) states that there exists a bijection of the sets of actions act $(A \times B, C) \cong$ act (B, C^A). The passage from $f : A \times B \to C$ to $f^\cdot : B \to C^A$, and conversely is accomplished with the help of two distinguished families of actions

$$(-) \xrightarrow{\lambda(-)} (A \times (-))^A \qquad\qquad A \times (-)^A \xrightarrow{\dot{\varepsilon}(-)} A$$

equally defined for any object (-) of the category. λ_B is the unique action corresponding to $1_{A \times B} : A \times B \to A \times B$ according to (1.8).††

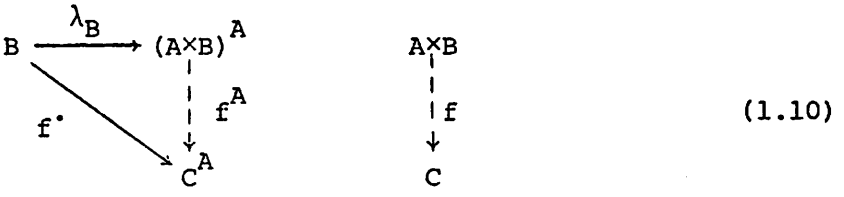
---

†Consequences of this axiom will be discussed in this paper with respect to Algol 68. But its importance goes beyond that. Consider the following example of lambda conversion from Lisp which involves the evaluation map: *evalquote ((lambda (xy)(cons (car x) y))((ab)(cd))) = (acd).*

††We do not bother to prove some details here. (1.8) states the existence of a certain adjunction and the interested reader is referred to [13], pp. 78-81.

For $b\varepsilon B$ $\lambda_B(b)$ is the action $A \to A\times B$ such that $\lambda_B(b)(a) = (a,b)$ $\forall\, a\varepsilon A$. On the other hand, given an action $f : A \to C$ and an element $a\varepsilon A$ we have $\varepsilon_C(a,f) = f(a)$.

For $f : A\times B \to C$, denote by $f^A : (A\times B)^A \to C^A$ the action sending each $g : A \to A\times B$ to the composite $f\cdot g : A \to C$. Then the unique $f^\cdot : B \to C^A$ corresponding to f is defined as the composite $f^A\cdot\lambda_B$ according to the diagram below



(1.10)

Conversely, given $f^\cdot : B \to C^A$ denote with $1_A\times f^\cdot : A\times B \to A\times C^A$ the action $(a,b) \to (a,f^\cdot(b))$. Then the unique $f : A\times B \to C$ is defined as the composite $\varepsilon_C\cdot(1_A\times f^\cdot)$ in (1.9).

As mentioned before (1.8) is satisfied in Set. It also holds for Clatt and this was stated by Scott (cf. [19], p. 113). For M_1 and M_2 complete lattices, the set of all continuous functions $M_2^{M_1}$ is indeed a complete lattice (cf. [19], p. 112) with the partial order defined as

$$f \leq g \quad \text{iff} \quad f(x) \leq g(x) \quad \forall x\varepsilon M_1 \tag{1.11}$$

Least upper bound of a subset $F \subseteq M_2^{M_1}$ is defined pointwise as

$$(\sup F)(x) = \sup\{f(x) : f\varepsilon F\} \tag{1.12}$$

To prove that $M_2^{M_1}$ is a complete lattice, one has to prove that supF is indeed a continuous function which turns out to be true (cf. [19], p. 112). The lower unit of $M_2^{M_1}$ is a continuous function $\Omega : M_1 \to M_2$

such that $\Omega(x) = \bot$, $\forall x \in M_1$.[†] The proof of the continuity of evaluation is due to Scott (cf. [19], p. 113).[††] To establish that one has to appeal to the lemma (cf. [19], p. 105) which says that a function $h : A \times B \to C$ is continuous iff it is continuous pointwise, i.e., if for $S \subseteq A$ and $S' \subseteq B$ directed sets we have

$$\forall y \in B \qquad h(\sup S, y) = \sup\{h(x,y) : x \in S\}$$
$$\forall x \in A \qquad h(x, \sup S') = \sup\{h(x,y) : y \in S'\}$$

With this in mind $\varepsilon_C : (a,f) \to f(a)$ can be easily proved continuous. With f fixed this is obviously continuous since f is, and with a fixed we must have for a directed set $F \subseteq C^A$

$$\varepsilon_C(a, \sup F) = \sup\{\varepsilon_C(a,f) : f \in F)\}$$

i.e.

$$(\sup F)(a) = \sup\{f(a) : f \in F\}$$

which holds by (1.12). Also, if $f^\cdot$ in (1.9) is continuous, so is $1_A \times f^\cdot$.

λ_B as defined in Set can be also proved continuous. Firstly $\lambda_B(b)$ which is a function $A \to A \times B$ such that $\lambda_B(b)(a) = (a,b)$ is continuous (so it is indeed an element of the lattice $(A \times B)^A$) by the following argument: Denote $\lambda_B(b) = f \in (A \times B)^A$ and let $D \subseteq A$ (be directed) then $f(\sup D) = (\sup D, b) = \sup\{(a,b) : a \in D\} = \sup\{f(a) : a \in D\}$. λ_B is continuous for if $D' \subseteq B$ (is directed) then $\forall a \in A$ we have $\lambda_B(\sup D')(a) = (a, \sup D') = \sup\{(a,b) : b \in D'\} = \sup\{\lambda_B(b)(a) : b \in D'\} = \sup\{\lambda_B(b) : b \in D'\}(a)$

[†] The partial order on the function set $M_2^{M_1}$ introduced for complete lattices is like the usual partial order of partial functions where for two partial functions f and g we say $f \leq g$ iff f is less defined than g and equal to g whenever defined. Ω corresponds to the totally undefined function.

[††] As far as we could see that does not complete the proof that (1.8) holds. That's why we establish here that λ_B is also continuous.

i.e., $\lambda_B(\sup D') = \sup\{\lambda_B(b) : b\in D'\}$.

Finally, we would like to prove that the function $f^A : (A\times B)^A \to C^A$ defined for Set is continuous if f is. First of all, for $g \in (A\times B)^A$ $f^A(g) = f\cdot g \in C^A$ is continuous as a composition of continuous functions. For $F \subseteq (A\times B)^A$ directed and for every $a\in A$ we have $(f\cdot\sup F)(a) = f(\sup F)(a) = f(\sup\{g(a) : g\in F\})$. From F directed it follows that $\{g(a) : g\in F\}$ is directed also and by the continuity of f we have $f(\sup\{g(a) : g\in F\}) = \sup\{f\cdot g(a) : g\in F\} = (\sup\{f\cdot g : g\in F\})(a)$, i.e., $f^A(\sup F) = \sup\{f^A(g) : g\in F\}$.

The importance of (1.8) can not be overestimated. As pointed out before it is a mathematical version of the lambda conversion in Lisp and the lambda calculus; it will play an important role in the construction of the conditional action in certain categories, etc. And yet the classical partial function approach fails to satisfy this axiom or, equivalently, it does not hold in $\underline{Set}_*$.[†] The lambda conversion as expressed in (1.8) holds in $\underline{Set}_*$ with a different construction of product which does not satisfy (1.3). Denote this product with $\dot\times$. Then we have $M_1\dot\times M_2 = (M_1 \setminus *_{M_2}) \times (M_2 \setminus *_{M_2}) \cup \{*\}$. With this definition we would be able to establish the bijection act $(A\dot\times B,C) \cong$ act (B,C^A). In Pfn this construction amounts to the construction of the set of ordered pairs (ordinary cartesian product of sets).

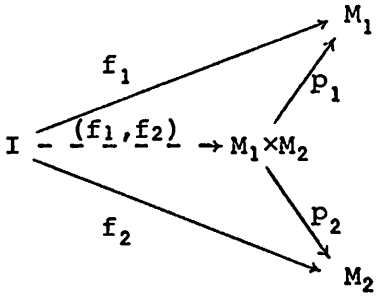
We conclude this section with the note, that the fifth rule of mode composition in Algol 68, which involves references, will be discussed in the last section, together with the storage model.

[†]For those familiar with some category theory we give a very quick proof by contradiction. Suppose (1.8) holds in $\underline{Set}_*$. Then $A\times(I+I) \cong A\times I + A\times I$. But $I + I \cong I$ and $A\times I \cong A$ so that we get $A \cong A + A$, an obvious contradiction.

2. Construction of Actions

What has been stated so far could be called an algebraic approach to structured data definition facilities of Algol 68. What makes this approach different from others is that the properties of "objects of structured data" were characterized in terms of actions of the abstract interpreter, so that the whole approach is a semantic one. We turn now to the construction (in the category) of some specific actions, most of them obtained by the elaboration of familiar programming phrases. These actions are constructed from the rules of data structuring as axioms.

We first discuss actions of Algol 68 related to structured values and values of united mode. Given two values $f_1 : I \to M_1$ and $f_2 : I \to M_2$ [†] of respective modes m1 and m2, the *structure display* (f_1,f_2) is a structured value of mode struct *(*m1 *s1,* m2 *s2)* defined uniquely by the property of $M_1 \times M_2$, determined in (1.3) as



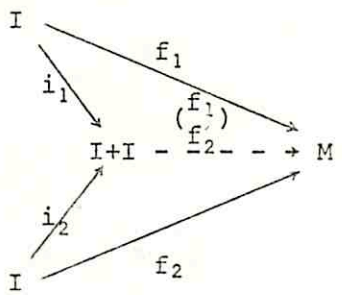
(2.1)

As noted before, if f_1 and f_2 are obtained by the elaboration of the phrases *e1* and *e2* respectively, the structured value (f_1,f_2) is externally denoted with the phrase *(e1,e2)*. On the other hand, if M is an object of mode struct *(*m1 *s1,* m2 *s2)* and given any structured value $f : I \to M$ corresponding to the phrase *e*, the elaboration of the

[†] Remember that in Set* we can not use I for this purpose. We'll use I freely in the rest of the paper, but the reader should bear this remark in mind.

phrase *si* of *e* gives rise to the composite action $p_i \cdot f$ and is in fact a value $I \to M_i$ of mode mi.

Dually, given values $f_1 : I \to M$ and $f_2 : I \to M$ of the same mode m, the *row display* $\binom{f_1}{f_2}$ is a value of mode []m, defined by (1.5) as a unique action such that



(2.2)

Now we see that the assumption (1.7) $N = I+I+\cdots$ makes a lot of sense, since $I+I$ is a subobject of N (denoted $I+I \rightarrowtail N$) and $I+I \to M$ therefore fits in the definition of one dimensional multiple values. We note that both structure display and row display generalize easily to arbitrary finite number of values f_i. If f_1 and f_2 are obtained by elaboration of the phrases *e1* and *e2* their row display is specified in the language by the phrase *(e1, e2)*. Row display and structure display are examples of collateral composition of actions.

Conditionals like $x \geq y$, $z < y$, etc., (we call them predicates, thinking equivalently of the characteristic function of the corresponding relation) are of course available in the language. We show that they fit in our theory as actions $M \to L$, yielding, therefore, either true or false. Operations like conjunction $\wedge$, as in $(x \leq y) \wedge (y < z)$, disjunction $\vee$, etc., come into the picture naturally, since we show that in a category with products, coproducts and exponentiation the set of

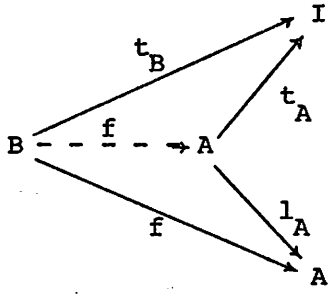
actions (predicates) act (M,L) where L=I+I carries the structure of a boolean algebra (for any object M). This conclusion follows from the rules of data structuring and since by (1.8) act (M,L) = L^M is again an object of the category, we are assured that predicates are values.

We first have to prove a lemma:

(2.3) For any object A of a category with products, coproducts and exponentiation there exists a canonical isomorphism

$$u : A \times L \to A+A$$

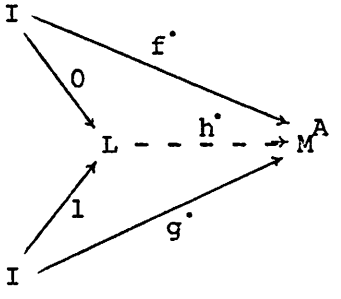
Recall that we denote this as $A \times L \cong A+A$. To prove the above statement remember that A+A is unique up to isomorphism. So all we have to show is that $A \times L$ has the universal property of A+A stated in (1.5). We first observe that $A \times I \cong A$ for any object A. This follows from the diagram below which shows that A has the universal property of the product of $A \times I$ as defined in (1.3) and since this determines $A \times I$ up to isomorphism we must have $A \times I \cong A$.



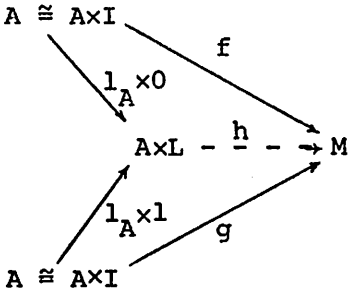
Now let $I \underset{1}{\overset{0}{\rightrightarrows}} L$ be the two injections. We'll show that

$$A \cong A \times I \underset{1_A \times 1}{\overset{1_A \times 0}{\rightrightarrows}} A \times L$$

are injections too.† For suppose we have a pair of actions $f : A\times I \to M$, $g : A\times I \to M$. Then by the exponentiation axiom (1.8) to f and g correspond unique $f^{\cdot} : I \to M^A$ and $g^{\cdot} : I \to M^A$. Now by the property of $L = I+I$ stated in (1.5) we conclude that there exists a unique $h^{\cdot} : L \to M^A$ such that



To this $h^{\cdot}$ corresponds a unique $h : A\times L \to M$. Moreover, with this $h^{\cdot}$ the diagrams



will commute, which proves that $A\times L$ satisfies (1.5) and so is isomorphic to $A+A$.

Now we prove the theorem:

(2.4) In a category with products, coproducts and exponentiation for any object A, act (A,L) carries the structure of a boolean algebra.

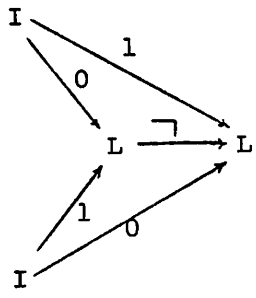
To prove (2.4) observe that $L = I+I$ is equipped with two injections $I \overset{0}{\underset{1}{\rightrightarrows}} L$. Define the action $\neg : L \to L$ according to (1.5) as the unique

---

†For $f : A \to C$ and $g : B \to D$, $f\times g : A\times B \to C\times D$ is the action $f\times g : (a,b) \to (f(a),f(b))$.

one such that the following diagram commutes

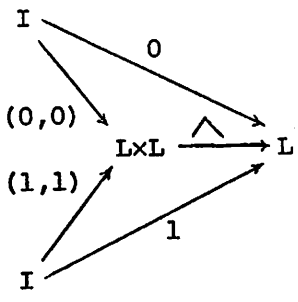
(2.5)

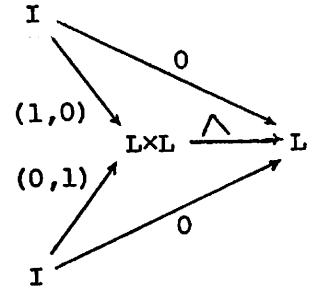


i.e., such that $\neg 0 = 1$ and $\neg 1 = 0$.[†] We thus get negation.

By (2.3) we have $L \times L \cong L + L$. By (1.3) (1,1), (0,1), (1,0), (0,0) are actions (structure displays): $I \to L \times L$. It is easy to see that these are the four injections $I \to (I+I) + (I+I)$. Now by (1.5) we can define $\wedge : L \times L \cong L + L \to L$ as a unique action which makes the following diagrams commutative.[††]

(2.6)

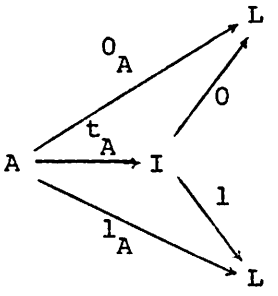




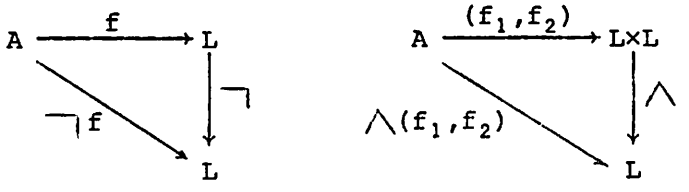
It can be verified that with the above definitions 0, 1, $\neg$ and $\wedge$ satisfy the equational laws of boolean algebras. This immediately implies that act (A,L) carries the structure of a boolean algebra with 0_A and 1_A defined as follows:

[†] Observe that $\neg$ is a value of mode proc (bool) bool.

[††] Observe that $\wedge$ is a value of mode proc (bool, bool) bool.

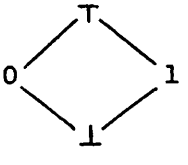


where t_A is a unique action $A \to I$. $\wedge$ and $\neg$ as defined above are the operations of conjunction and negation of the boolean algebra act (A,L) since we have



Given any $f : A \to B$ it can be verified that $L^f : L^B \to L^A$, sending each $g : B \to L$ into the composite $A \xrightarrow{f} B \xrightarrow{g} L$ is a boolean algebra homomorphism, but that's of little interest to us here.

This discussion applies entirely to Set, but in Clatt and Set_* we have some problems. Firstly, $A \times L \cong A+A$ is not true anymore in Clatt, since $+$ as we defined it for Clatt does not satisfy (1.5). However, it is still possible to define continuous operations $\neg : L \to L$ and $\wedge : L \times L \to L$[†] such that (2.5) and (2.6) commute. So for example $\neg$ is defined as $\neg(0) = 1$, $\neg(1) = 0$, $\neg(\top) = \top$, $\neg(\bot) = \bot$ which is continuous since $L = I+I$ is of the form



[†]Basically, this is possible since we can define an intuitively natural continuous function $u : L \times L \to L+L$ as will be explained later. A similar conclusion is true for Set_*.

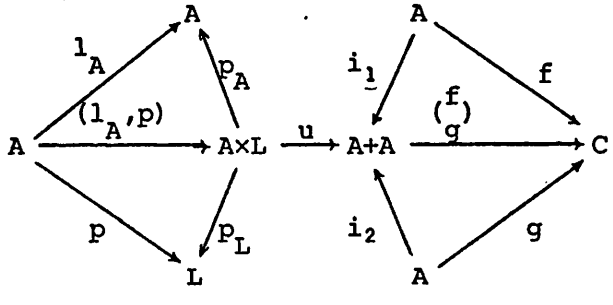
0 corresponding to <u>false</u> and 1 to <u>true</u>.

We remark that in this section we are using some of the machinery and results of categorical algebra (cf. [11] and [21]).

Now we can look carefully at conditional actions. A *conditional action* is expressed as <u>if</u> *boolean* <u>then</u> *expression1* <u>else</u> *expression2* <u>fi</u>. Suppose that the elaboration of *boolean* gives rise to the predicate action $p : A \to L$ and of *expression1* and *expression2* to actions $A \overset{f}{\underset{g}{\rightrightarrows}} C$ respectively. The semantics of the above conditional action is well known: *boolean* is elaborated first; if it yields the value <u>true</u> (<u>false</u>) *expression1* (*expression2*) is elaborated. We construct the action corresponding to the overall effect of the conditional action using the axioms introduced so far. So we prove:

(2.7) There exists a conditional action in any category with binary products and coproducts, and the exponentiation.

Recall that in a category satisfying the above requirements there exists a canonical isomorphism $u : A \times L \to A+A$ by (2.3). The desired conditional action is the composite $\binom{f}{g} u(1_A, p)$ in the diagram

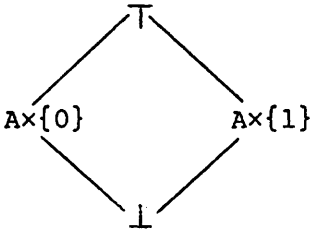


Observe that in a more general case we would have $f : A \to C$ and $g : A \to D$ so that the conditional action is going to be

$$\begin{pmatrix} i_C \cdot f \\ i_D \cdot g \end{pmatrix} u(1_A, p) \; : \; A \to C+D$$

where $i_C : C \to C+D$ and $i_D : D \to C+D$ are injections.

In Set the above construction works, but not quite so nicely in Clatt. $A \times L \cong A+A$ is not true in Clatt, but it is possible to construct a continuous function $u : A \times L \to A+A$ sending disjoint copies $A \times \{0\}$ and $A \times \{1\}$ which are contained in $A \times L$ to the corresponding copies of $A+A$ where we think of $A+A$ as



To prove this, we have to rely on the previously mentioned fact that u is continuous in both arguments iff it is continuous in each argument separately. The desired u is defined as $u(x,0) = (x,0)$, $u(x,1) = (x,1)$, $u(x,\top) = \top$, $u(x,\bot) = \bot$. A straightforward checking shows that u defined this way is indeed continuous and the conditional action similar to (2.7) can be recaptured with the proper intuition to support it.

We can follow the same pattern in $\underline{Set}_*$ and construct a conditional action according to (2.7) except that u again would not be an isomorphism. Still it would be a very natural map such that the construction of the conditional action obtained this way has the desired semantics. Observe that $A \times 1 = A \times \{1\} \cup A \times \{0\} \cup A \times \{*_1\}$ and the base point preserving function just adjoints two (disjoint) sets $A \times \{1\}$ and $A \times \{0\}$ identifying points $(*_A,1)$ and $(*_A,0)$ which becomes a new base point of $A+A$ and sends $A \times \{*_1\}$ to that base point.

The most critical action for any kind of algebraic theory of programming is certainly the repetitive action.† The *repetitive action* in

†As is well known this action in general cannot be constructed in Set. Because of the nonhalting situations it is in general a partial function.

its full generality is represented by the phrase:

<u>for</u> *i* <u>from</u> *start* <u>by</u> *step* <u>to</u> *stop* <u>while</u> *condition* <u>do</u> *doclause*

We consider the option of the form:

<u>while</u> *condition* <u>do</u> *doclause*

Suppose that *condition* elaborates to the predicate action $p : X \to L$ and *doclause* to the action $f : X \to X$. To capture the iterative nature of the repetitive action in <u>Clatt</u> we construct a distinguished action $k : X \to X$ such that

$$\begin{array}{ccccc} X & \xrightarrow{(1_X,p)} & X\times L & \xrightarrow{u} & X+X \\ & \searrow^{k} & & & \downarrow{\scriptstyle \begin{pmatrix} k\cdot f \\ 1_X \end{pmatrix}} \\ & & & & X \end{array} \qquad (2.8)$$

This distinguished solution of the equation $k = \begin{pmatrix} k\cdot f \\ 1_X \end{pmatrix} u(1_X,p)$ is the repetitive action and corresponds to the elaboration of the phrase <u>while</u> *condition* <u>do</u> *doclause*. There is in general more than one continuous function k yielding a commutative diagram in (2.8). Similarly to the situation we had for <u>Clatt</u> with (1.5) it is natural to choose the least k satisfying the equation $k = \begin{pmatrix} k\cdot f \\ 1_X \end{pmatrix} u(1_X,p)$. The desired repetitive action is established as the minimal fixed point of the continuous function

$$\phi : X^X \to X^X$$

sending a continuous function $k \in X^X$ to the continuous function $\begin{pmatrix} k\cdot f \\ 1_X \end{pmatrix} u(1_X,p)$. The existence of a fixed point is guaranteed for every monotone function on a complete lattice by the Knaster-Tarski theorem (cf. [18] and [19]) and for our continuous ϕ the minimal fixed point is calculated as

$$\sup\{\phi^n(\Omega)\}_{n=0}^{\infty}$$

where Ω is the lower unit of the lattice X^X (totally undefined function). To prove that ϕ is continuous let $F \subseteq X^X$ (be directed). We have to show

$$\left[\begin{matrix}(\sup F)\cdot f\\ 1_X\end{matrix}\right]u(1_X,p) = \sup\{\left[\begin{matrix}y\cdot f\\ 1_X\end{matrix}\right]u(1_X,p) : y\varepsilon F\}, \quad \text{i.e.,} \quad \forall x\varepsilon X$$

$$\left[\begin{matrix}(\sup F)\cdot f\\ 1_X\end{matrix}\right]u(1_X,p)(x) = \sup\{\left[\begin{matrix}y\cdot f\\ 1_X\end{matrix}\right]u(1_X,p) : y\varepsilon F\}(x)$$

$$= \sup\{\left[\begin{matrix}y\cdot f\\ 1_X\end{matrix}\right]u(1_X,p)(x) : y\varepsilon F\}$$

Denote $x' = u(1_X,p)(x) \varepsilon X+X$ and we have to prove

$$\left[\begin{matrix}(\sup F)\cdot f\\ 1_X\end{matrix}\right](x') = \sup\{\left[\begin{matrix}y\cdot f\\ 1_X\end{matrix}\right](x') : y\varepsilon F\}$$

Recalling the weak (coproduct) construction of (1.5) for Clatt and observing that

$$(\sup F)(f(x')) = \sup\{y(f(x')) : y\varepsilon F\}$$

holds by (1.12) it is easy to see that the desired equation holds.

(2.8) can also be used as the basis for the construction of the repetitive action in $\underline{Set}_*$. k is a base point preserving function $X \to X$ such that $k(x) = f^n(x)$ if $p\cdot f^n(x)$ evaluates to false for some finite n and $k(x) = *_x$ if not.

3. Storage Model

Any discussion of further actions of the language would be impossible without introducing the storage model. Only after we give a model of run-time representation of the language, can we talk about identity declarations,

assignments and procedure calls, thus completing our global algebraic discussion of Algol 68.† But before discussing the storage model we have to deal with one more rule for the composition of modes.

If m is a mode, then ref m is another mode. Values of mode ref m refer to values of mode m and are therefore called references or names. While identifiers are external accesses to values, names are *internal* accesses to values.†† Being internal (in our approach, living in the category) they are values themselves. This powerful rule of data structuring is crucial for Algol 68. We have already seen an example (the declaration of the mode s-expr) of using references in building list-like data structures.

Moreover, introducing references as values, one can easily achieve what is usually termed as a call by reference. Indeed, a value of mode proc (int, ref [] char) bool is a routine with the second parameter being called by reference. When this procedure is called, only a reference to a row of characters will be given to the procedure as the second actual parameter, instead of copying the whole array on the stack.

Following the strategy in the first section we would like to establish an axiom which says how to construct, from an object M of mode m, the object $\bar{M}$ of mode ref m. It seems reasonable to start from the assumed

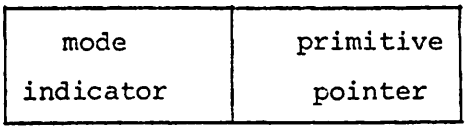
---

†In addition to stack controlled values Algol 68 features values which do not fit in the last-in first-out principle of a stack. Values of this type are stored in a random organized memory area called the *heap*. We do not deal here with the heap part of the structure of the language.

††David Stemple pointed out that the terminology should be just the other way around. An identifier does not identify the value, it only names it, since there may be many identifiers possessing the same value, and the same identifier in different ranges possessing different values. However, a name identifies the value completely; it is the name that should be called the identifier.

object $\bar{P}$ of *primitive references* (pointers). In constructing $\bar{M}$ of mode ref m we use $\bar{P}$ and (possibly) some of the already established constructions (for example product, coproduct, etc.) so that as the result we obtain an object $\bar{M}$ of the category.† A distinguished action $\bar{M} \to M$ or $\bar{M} \to I$ is called the *dereference action*. Given a reference $I \to \bar{M}$ we apply the dereference action (or, compose it with the dereference action) to obtain the (possibly unspecified) value it refers to.

One natural way of constructing references is represented in the figure below



According to the above figure a reference is a pair (mode indicator, primitive pointer). In certain categories (for example Set, Clatt, etc.) this amounts to the statement that if M is an object of mode m then the object $\bar{M}$ of mode ref m is $\bar{M} \cong \bar{P} \times I$. The existence of the isomorphism $\bar{P} \times I \cong \bar{P}$ just says that in principle there is not much difference between having a primitive mode reference, and for every mode m a data structuring rule for constructing the mode ref m. Algol 68 chooses the second alternative and it should be clear that it gives rise to more efficient implementation (type checking is easier). It is interesting to observe that if in $\underline{Set}_*$ we construct $\bar{M}$ as $\bar{P} \times 1$ no longer is $\bar{P} \times 1 \cong \bar{P}$ but $\bar{P} \dot{\times} 1 \cong \bar{P}$. For a number of practical aspects of implementing references we refer the reader to [4] and [5].

---

†We are by no means satisfied with the fact that the passage from the mode m to the mode ref m does not seem to correspond to a nice construction similar to (1.3), (1.5), (1.8), etc. We leave this as an open problem.

In representing an activation record† of a block (or more generally a range) we base our approach on the observation that such a record for a block in which n identifiers *s1*, ..., *sn* of respective modes m1, ..., mn are declared, can be conceptually viewed as a structured value of mode struct (m1 *s1*, ..., m2 *s2*). If m1, ..., mn denote objects M_1, ..., M_n, then this structure display corresponds to an action (value)

$$I \rightarrow M_1 \times \cdots \times M_n$$

of the category. However, the value possessed by an identifier can be a name, which refers to some other value, possibly unspecified at certain points of the execution, or a constant. To illustrate this, consider the general form of the identity declaration

$$\underline{m}\ s = e \qquad (3.1)$$

where m specifies a mode (and is called a declarer), *s* is an identifier and *e* is an expression which, upon elaboration, yields a value of mode m. Declarations which make an identifier possess a constant value, which can therefore be changed only by redeclaring it, are for example

```
real x = 7.45;
proc f = (real a, real b) real : (a+b)/2 - sqrt(axb);
```

We can never change the values possessed by the identifiers *x* and *f* in the above declarations by assignations, calls, etc. On the other hand the declarations

```
real x := 7.45;
proc f := (real a, real b) real : (a+b)/2 - sqrt(axb);
```

†A block activation record is the local storage area associated with every instance of execution of the block.

which are the abbreviated forms of:

$$\underline{\text{ref real}}\ x = \underline{\text{loc real}}\ :=\ 7.45;$$

$$\underline{\text{ref proc}}\ (\underline{\text{real}},\ \underline{\text{real}})\ \underline{\text{real}} = \underline{\text{loc proc}}\ (\underline{\text{real}},\ \underline{\text{real}})\ \underline{\text{real}}\ :=$$

$$(\underline{\text{real}}\ a,\ \underline{\text{real}}\ b)\ \underline{\text{real}}\ :\ (a+b)/2 - sqrt(a \times b);$$

cause quite a different effect. Now the values possessed by the identifiers x and f are names, which refer respectively to the internal representations of 7.45 and (real a, real b) real : $(a+b)/2 - sqrt(a \times b)$. Elaboration of the phrases loc real and loc proc (real, real) real creates the names, i.e., values of respective modes ref real and ref proc (real, real) real. Now we can assign to f and x. If we do so, the names possessed by them won't change, but the values referred to by these names will. Observe that this subtle mechanism distinguishes between constant and variable procedures.

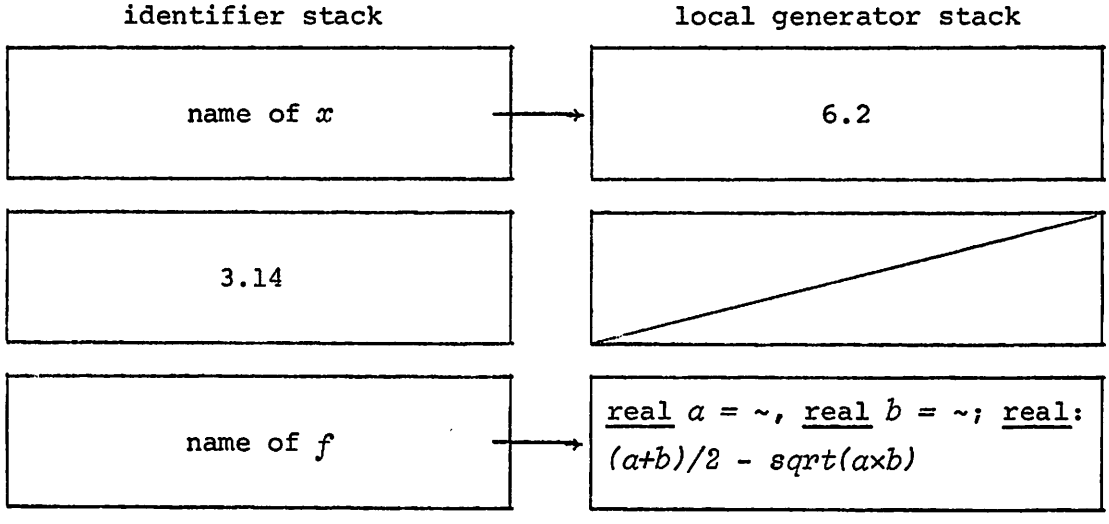
To capture this subtlety we adopt the approach based on [4] that the storage model consists of two stacks, called the *identifier stack* and the *local generator stack*. When actually implementing the language, both stacks can be realized as a single stack (including the working part of it for evaluating expressions) called the *range stack*. But in our conceptual view, we'll use the two stack model. The stacks are going to be represented using products.

The idea of the two stack model is the following. Values possessed by identifiers will be placed on the identifier stack at fixed relative positions from the beginning of the block activation record. Since those values can be names, the values they refer to will be placed at appropriate places on the local generator stack. For example, the activation record

of the block

```
begin real x := 6.2;
      real pi = 3.14;
      proc f := (real a, real b) real:                    (3.2)
      (a+b)/2 - sqrt(a×b);
end
```

will be



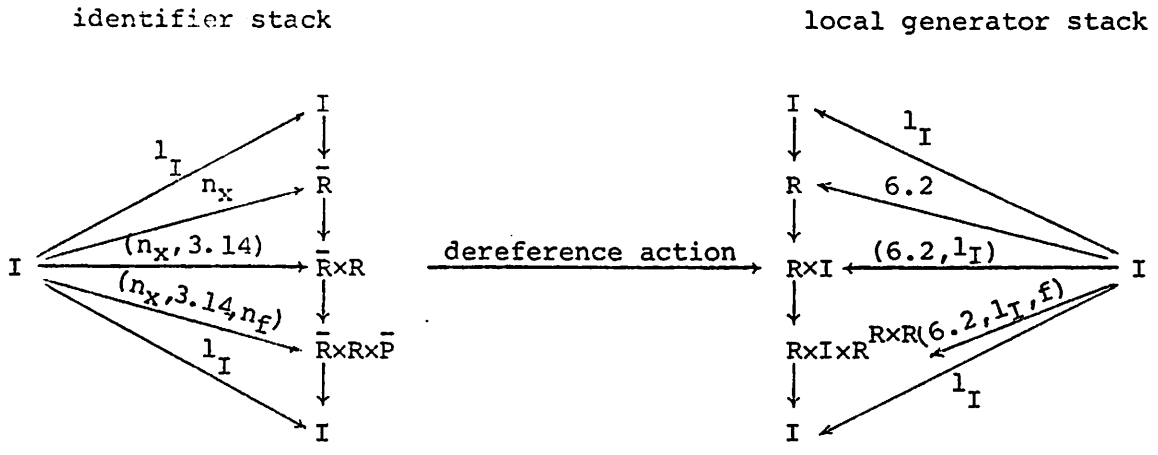
(3.3)

Note that we denoted externally the internal representation of the routine possessed by f. The mathematics is a shorthand indeed, since if we denote with $\bar{P}$ the object of mode ref proc (real, real) real and with $\bar{R}$ the object of mode ref real, the above activation record can be represented as two structure displays

$$I \longrightarrow \bar{R}\times\bar{R}\times\bar{P} \qquad\qquad R\times I \vee R^{R\times R} \longleftarrow I$$

Every block is included in some outer range and the whole program is in the range of the standard environment of the language (called the standard prelude). Forgetting for simplicity the outer range and denoting with n_x and n_f the names possessed by x and f, we can represent the

sequence of actions of the abstract interpreter (extensions and contractions of the two stacks) as follows:



(3.4)

Observe that in the above example all declarations are initialized, i.e., they include assignments. Usually, these two effects are separated. Identity declarations are actions on the identifier stack. Depending upon the mode m in (3.1) declarations create constants or names on the stack. A declaration of the form

$$\underline{\text{real}}\ x$$

is an action

$$I \to \bar{R}$$

of the category. There may be many actions $I \to M$ of the category, but only one $M \to I$, with the semantics of the exit from a range (deallocating storage), as is illustrated in the above diagrams. Observe that the effect of the declaration

$$[1 : 3]\ \underline{\text{real}}\ x$$

is the creation of a name, and not allocation of (much more) storage *for*

the whole array. The latter is going to take place only at the time of assignment.

Declarations are the only actions on the identifier stack. All other actions, assignations and calls in particular, act on the local generator stack. This implies that we cannot assign to identifiers possessing constant procedures. On the other hand, variable procedures which have a name and therefore the value it refers to can be changed acting on the local generator stack.

After this general discussion, we can be more precise about assignment actions and procedure calls. Let n be a phrase which upon elaboration gives a name referring to a value of mode m. Let e be a phrase which, when elaborated, gives a value of mode m. Then an *assignment action* is obtained by the elaboration of the phrase

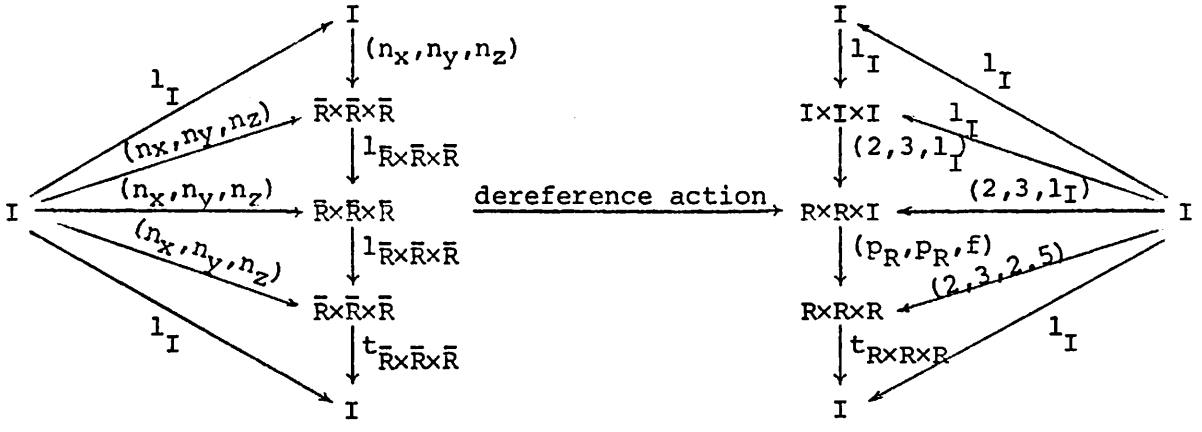
$$n := e$$

which makes the name yielded by the elaboration of n refer to the value obtained by the elaboration of e. For example:

```
begin real x, y, z;
      (x := 2, y := 3); ←
      z := (x+y)/2 ←                    (3.5)
end
```

Denote the names possessed by x, y and z respectively with n_x, n_y and n_z. Here is the trace of actions of the interpreter, which shows clearly that assignments are actions on the local generator stack.

identifier stack local generator stack



(3.6)

In the above diagrams, f denotes the function $f(x,y) = (x+y)/2$ and p_R projection. Observe that $I\times I\times I \cong I$ and $R\times R\times I \cong R\times R$ as proved before. We discuss only what happens between the two points in (3.5) indicated by the arrows. At the first point, there exist three names, n_x, n_y, n_z. The first two refer to respective values 2 and 3, obtained by dereferencing n_x and n_y respectively. Dereferencing n_z gives 1_I, the only value of I, which is interpreted as the unspecified value, since $R\times R\times I \cong R\times R$. Assignment $z := f(x,y)$ gives rise to the action on the local generator stack

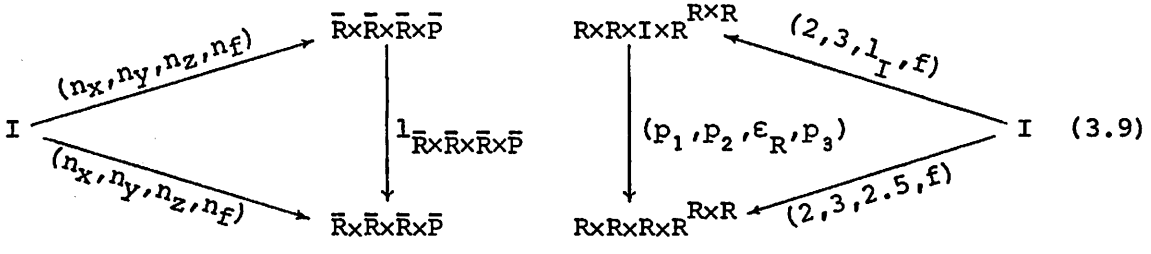
$$R\times R \xrightarrow{(p_R,p_R,f)} R\times R\times R \qquad (3.7)$$

The effect of this action is that if we dereference n_z at the point corresponding to the second arrow, we see that the value it refers to is $(2+3)/2 = 2.5$. Observe also that given two projections $p_R : R\times R \to R$ and $f : R\times R \to R$, $(p_R,p_R,f) : R\times R \to R\times R\times R$ is the unique action to the product defined in (1.3).

We conclude the section with the discussion of procedure calls and their side-effects. The evaluation action defined in (1.8) will play an important role. Consider the following example in which a variable procedure is declared:

```
begin real x, y, z;
    (x := 2, y := 3);
proc f := (real a, real b) real : (a+b)/2;          (3.8)
    z := f(x, y);
end
```

We concentrate only on the changes of the environment caused by the call in the assignment $z := f(x, y)$. Denoting the names possessed by x, y z and f with n_x, n_y, n_z and n_f respectively we represent this change as



(3.9)

In the above diagrams p_1, p_2 and p_3 denote the three projections from the product $R\times R\times R^{R\times R}$. The call $z := f(x, y)$ gives rise to the action

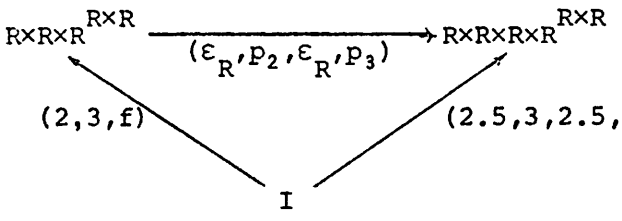
$$R\times R\times R^{R\times R} \xrightarrow[(p_1,p_2,\varepsilon_R,p_3)]{} R\times R\times R\times R^{R\times R} \qquad (3.10)$$

where $\varepsilon_R : R\times R\times R^{R\times R} \to R$ is the evaluation from (1.8).

The procedure f was such that it does not cause side effects. Suppose now that f in (3.8) is declared as

```
proc f := (real a, real b) real : x := (a+b)/2;
```

The call in $z := f(x, y)$ will cause the side effect of changing the value possessed by x and will give rise to the action



(3.11)

Concluding this section we'll make some comments on recursive procedures. There exists a standard way of implementing (reentrantly) recursive procedures in block structure languages. Each recursive call of such a procedure causes a new activation record to be placed on the range stack (cf. [23], p. 275). Observe that with the algebraic language developed in this paper there are no problems in treating recursive procedures in the same manner. In Clatt this implementation-oriented approach has a conceptual alternative. Consider for example the following declaration:

$$\underline{\text{proc}}\ f = (\underline{\text{int}}\ x)\ \underline{\text{int}} : \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x \times f(x-1)\ \underline{\text{fi}};$$

The internal representation of the above phrase is a function $N \to N$ which is in Clatt obtained as the least upper bound of the sequence of continuous functions corresponding to $f0, f1, f2, \ldots$ where

$$\underline{\text{proc}}\ fi = (\underline{\text{int}}\ x)\ \underline{\text{int}} : \underline{\text{if}}\ x \leq i\ \underline{\text{then}}\ x!\ \underline{\text{else}}\ \underline{\text{skip}}\ \underline{\text{fi}};^{\dagger}$$

This least upper bound is clearly the factorial function. For certain kinds of recursive procedures the way in which the language interpreter is implemented becomes essential if the function it computes is to be the

---

†We want skip in this context to deliver an undefined value of the required mode, so that it internally corresponds to the lower unit $\perp$ of the lattice.

same as the minimal fixed point of a recursive function definition. For more details the reader is referred to [15] and [22].

—•—

In this concluding section we summarize briefly the discussion presented in the paper.

Current trends in structured programming (cf. [7]) emphasize clarity and intellectual manageability of programming solutions. We argue that any mathematical theory of programming should follow the same trend. That's why we advocate the use of algebraic tools as the most elegant ones that we know of.

We presented a careful algebraic analysis of some of the structure of Algol 68. When viewed in the appropriate way the apparently complicated structure of the language turns out to be very elegant and interesting. We formulated a collection of algebraic constructions that axiomatize the mode composition rules of Algol 68 and showed that these rules of data structuring are all that should be introduced by definition. Actions corresponding to standard programming phrases should be constructed from these axioms and the storage model developed using these rules of data structuring only. We hope that this interesting "closure" property of Algol 68 is an insight into the language.

Unfortunately, we have not been able to produce a category that satisfies strictly the ideal set of axioms needed for Algol 68. But we showed to what extent Scott's lattice-theoretic approach and the classical partial function approach are relevant to the problem. We analyzed how close these two models get to a category ideal for our purposes.

Even though Scott's theory is relevant to the particular language we were interested in, we pointed to some problems of that theory in capturing united modes of Algol 68, construction of conditional actions, etc. A careful comparison of the two approaches, the one in which sets of values (domains) are simply sets or pointed sets and the other one in which they are turned into complete lattices, is given. It was shown what the gains and the losses are in this passage.

It is important to remark that Algol 68 should not be blamed for the fact that neither the lattice-theoretic approach nor the conventional partial function approach are ideal models. As pointed out briefly, in some other languages with nice structure (Lisp, for example) we would encounter similar problems.

The importance of investigating the relevance of Scott's theory to real programming languages (we have chosen Algol 68) we see in the recent unified approach (relying on Scott's theory) to different techniques for proving correctness, equivalence and termination of programs (cf. [14], [15] and [22]).

There are some minor points to the theory of semantics brought up by our discussion. Structured values are nothing else but the Vienna objects (cf. [16] and [24]) and our approach to formal storage model using products is essentially an algebraic version of some of the Vienna definition language approach. We have not investigated this carefully, but there is probably more to the relationship between the interpretive style of semantic definition (the Vienna group) and Scott's mathematical theory of semantics.

Finally, Algol 68 turns out to be a language which makes explicit some algebraic aspects in the structure of programming language. That explains our choice of a language for the approach presented in this paper.

References

1) J. C. Abbott, Sets, Lattices and Boolean Algebras, Allyn and Bacon, Boston, 1969.

2) H. Bekić and K. Walk, Formalization of Storage Properties, in Symposium on Semantics of Algorithmic Languages, edited by E. Engeler, Lecture Notes in Mathematics, No. 188, Springer-Verlag, New York, 1971.

3) P. Branguart, J. Lewi, M. Sintzoff, and P. L. Wodon, The Composition of Semantics in Algol 68, Communications of A.C.M., Vol. 14, No. 11, November, 1971.

4) P. Branguart and J. Lewi, A Scheme of Storage Allocation and Garbage Collection, in Algol 68 Implementation, Proceedings of the IFIP Conference on Algol 68 Implementation, edited by J. E. L. Peck, Worth-Holland, 1971.

5) P. Branguart and J. Lewi, On the Implementation of Local Names in Algol 68, MBLE Research Laboratory Report R121, Brussels, September, 1970.

6) D. M. Berry, Block Structure: Retention or Deletion?, Proceedings of Fourth Annual Symposium on Theory of Computing, 1972.

7) E. W. Dijkstra, The Humble Programmer, Communications of the ACM, Vol. 15, No. 10, October, 1972.

8) J. A. Goguen, Jr., On Homomorphisms, Simulations, Correctness and Subroutines for Programs and Program Schemes, Computer Science Dept., University of California, Los Angeles, 1972.

9) C. A. R. Hoare, Notes on Data Structuring, in O. J. Dahl et al., Structured Programming, Academic Press, 1972.

10) F. W. Lawvere, An Elementary Theory of the Cátegory of Sets, mimeographed notes, University of Chicago; condensed version in Proc. Nat. Acad. of Sci. 52 (1964).

11) F. W. Lawvere, Theories as Categories and the Completeness Theorem, Journal of Symbolic Logic 32 (1967).

12) C. H. Lindsey and S. G. van der Meulen, Informal Introduction to Algol 68, North-Holland, Amsterdam, 1971.

13) S. Mac Lane, Categories for the Working Mathematician, Springer-Verlag, New York, 1972.

14) Z. Manna and J. Vuillemin, Fixpoint Approach to the Theory of Computation, Communications of the ACM, Vol. 15, No. 7, July, 1972.

15) Z. Manna, S. Ness, and J. Vuillemin, Inductive Methods for Proving Properties of Programs, Communications of the A.C.M., Vol. 16, No. 8, August, 1973.

16) E. J. Neuhold, The Formal Description of Programming Languages, IBM Systems Journal, No. 2, 1971.

17) J. C. Reynolds, Notes on a Lattice-Theoretic Approach to the Theory of Computation, Systems and Information Science, Syracuse University, October, 1972.

18) D. Scott, The Lattice of Flow Diagrams, in Symposium on Semantics of Algorithmic Languages, edited by E. Engeler, Lecture Notes in Mathematics, No. 188, Springer-Verlag, New York, 1971.

19) D. Scott, Continuous Lattices, Toposes and Algebraic Geometry, Lecture Notes in Mathematics No. 274, Springer-Verlag, 1971.

20) A. van Wijngaarden et. al., Report on the Algorithmic Language Algol 68, Numerische Mathematik, 14 (1969), Springer-Verlag.

21) H. Volger, Logical Categories, Dalhousie University, Halifax, March, 1971.

22) J. Vuillemin, Correct and Optimal Implementation of Recursion in a Simple Programming Language, Proceedings of Fifth Annual ACM Symposium on Theory of Computing, Austin, May, 1973.

23) P. Wegner, Programming Languages, Information Structures and Machine Organization, McGraw-Hill, New York, 1968.

24) P. Wegner, The Vienna Definition Language, Computing Surveys, Vol. 4, No. 1, March, 1972.

COMPUTER AND INFORMATION SCIENCE

TECHNICAL REPORTS

The following TECHNICAL REPORTS are now available at the Computer and Information Science Department, Graduate Research Center.

70C-2 Organizational Principles for Embryological and Neurophysiological Processes by Michael A. Arbib.

70C-3 On the Likely Evolution of Communicating Intelligence on Other Planets by Michael A. Arbib (August 1, 1970, revised December 30, 1970.

70C-4 Contextual Error Detection by Roger W. Ehrich and Edward M. Riseman.

70C-5 Transformations and Somatotopy in Perceiving Systems by Michael A. Arbib.

70C-6 Organizational Principles for Theoretical Neurophysiology by Michael A. Arbib, (August 15, 1971).

71B-1 The Definition and Validation of the Radix Sorting Technique by John A. N. Lee, (January, 1972).

71B-2 Two Papers on Group Machines by Michael A. Arbib.

71B-4 Automata with Ranked State Sets by Dieter Schütt.

71C-6 Machines in a Category by M. A. Arbib and E. G. Manes.

72A-1 A Study of the Constraints upon the Parallel Dispatching and Execution of Machine Code Instructions by Caxton C. Foster and Edward M. Riseman.

72B-1 The Formal Definition of the Basic Language by John A. N. Lee.

72B-2 Decomposable Machines and Simple Recursion by Michael A. Arbib and E. Manes.

72C-1 A Contextual Postprocessing System for Error Detection and Correction in Character Recognition by Edward M. Riseman and A. Hanson (October 1972).

73B-1 Adjoint Machines, State-Behavior Machines, and Duality by Michael A. Arbib and Ernest G. Manes (January 1973).

73B-2 Natural State Transformations by Suad Alagić (February 1973). (Revised Nov.1973)

73C-3 A Model of Posited Decisionary and Learning Mechanisms in Mammalian CA-3 Hippocampus by William Kilmer, T. McLardy, and M. Olinski (February 1973).

73C-4 Four Faces of Hal: Using Artificial Intelligence Techniques in Computer-Assisted Instruction by Howard A. Peelle and Edward M. Riseman (March, 1973).

73C-5 System Design of an Integrated Pattern Recognition System, or How to Get the Best Mileage out of your Used Pattern Classifier by A.R. Hanson and E.M. Riseman, (June 1973).

73C-6 Neural Models of Spatial Perception and the Control of Movement, by Michael A. Arbib, C. Curt Boylls & Parvati Dev (June 1973).

73B-3 Foundations of System Theory, I by Michael A. Arbib and Ernest G. Manes, (July, 1973.)

73A-1 ULD and a Description of the PDP-8, by John A. N. Lee (September 1973).

73B-4 Time-Varying Systems, by Michael A. Arbib and Ernest G. Manes (November 1973).

73C-7 Model of a Plausible Learning Scheme for CA3 Hippocampus, by William Kilmer and Melanie Olinski, (Nov., 1973).

73B-5 Algebraic Aspects of Algol 68, by Suad Alagić (November 1973).

73C-8 Biology of Decisionary and Learning Mechanisms in Mammalian CA3-Hippocampus, by William Kilmer (November 1973).

COMPUTER AND INFORMATION SCIENCE

TECHNICAL NOTES

The following TECHNICAL NOTES are now available at the Computer and Information Science Department, Graduate Research Center.

TN/CS/00001 (Out of Date) — Current Research Toward the Standardization and Formal Definition of PL/I by John A.N. Lee (April 1,1968).

TN/CS/00002 — Discrete Markov Chains: An Heuristic Approach by Sue N. Stidham (July 1,1968).

TN/CS/00003 — The Domelki Syntactic Analysis Algorithm by Susan L. Gerhart (August 28,1968).

TN/CS/00004 — A Survey of Hashing Techniques by John A.N. Lee (September 15,1968).

TN/CS/00005 — The Recognition and Use of Null Elements in a Syntax Directed Translator by John A.N. Lee (September 19,1968).

TN/CS/00006 (Revision is TN/CS/00020) — SYNFUL, A Proposed General Purpose Translator System by John A.N. Lee (October 1,1968).

TN/CS/00007 — Sorting Almost Ordered Arrays by Caxton C. Foster (November 18,1968).

TN/CS/00008 (Out of Date) — Vienna Definition Language--Semantics by John A.N. Lee (December 13,1968).

TN/CS/00009 — An Examination of Two Hash Transforms by Caxton C. Foster (May 9,1969).

TN/CS/00010 — Multiplexing Without Tears by Caxton C. Foster (May 30,1969).

TN/CS/00011 (Out of Date) — Vienna Definition Language--A Generalization of Instruction Definitions by John A.N. Lee and Delmore Wu (April,1969).

TN/CS/00012 — An Unclever Time-Sharing System by Caxton C. Foster (October,1969).

TN/CS/00013 — The Formal Definition of the Basic Language by John A.N. Lee (April,1970). (Also published in Computer Journal and as Technical Report 72B-1)

TN/CS/00014 — A Debugging Aid by Caxton C. Foster and Hugh C. Schulz (January 16,1970).

TN/CS/00015 — Conditional Interpretation of Operation Codes by Caxton C. Foster and Robert H. Gonter (February,1970).

TN/CS/00016 — Some Simple Algorithms for Content Addressable Memories by Caxton C. Foster (July,1970).

TN/CS/00017 — A Data Distributor--The Sprinkler System by Caxton C. Foster (July,1970).

TN/CS/00018 An Annotated Bibliography on Syntax-Directed Translation by John A.N. Lee, Taveta K. Bogert, and Helen Gigley (July,1970).

TN/CS/00019 Chargoggaggoggmanchaugagoggchaubunagungamaug--A Novel Multiply-by-Three Circuit by Caxton C. Foster, Edward Riseman, Fred Stockton, and Conrad Wogrin (September,1970).

TN/CS/00020 SYNFUL--A General Purpose Translator System and Extensive Modification of Technical Note #00006 by John A.N. Lee and Helen Gigley, edited by Taveta K. Bogert (November,1970).

TN/CS/00021 Maintenance Manual for the UMASS Timesharing Version of SYNFUL by Ronald Lautmann (November,1970).

TN/CS/00022 Microprogramming: A Design Alternative by Michael J. Sullivan (December,1970).

TN/CS/00023 A Simulated Associative Memory by Caxton C. Foster (December,1970).

TN/CS/00024 A Five Tape Algorithm for the Instant Playback Problem by Caxton C. Foster (December,1970).

TN/CS/00025 A Comparison of Simulation Languages by Albert W. Zukatis (January,1971).

TN/CS/00026 A Stack Oriented Computer by Caxton C. Foster (April,1971).

TN/CS/00027 Certain Formal Properties of the Vienna Definition Language by John A.N. Lee.

TN/CS/00028 When the Chips are Down by Caxton C. Foster (January,1972).

TN/CS/00029 RAUPEDATA-11 -- A Sophisticated Debugging Program for the PDP-11 by Edward G. Fisher (February,1972).

TN/CS/00030 A Tutorial on Cobol Extensions to Handle Data Bases: The Data Base Group Report by Robert W. Taylor (February,1972).

TN/CS/00031 System Design: Process Models by Richard H. Eckhouse, Jr. (April,1972).

TN/CS/00032 Automated Accounting Systems by Richard H. Eckhosue, Jr. (April,1972).

TN/CS/00033 A Generalization of AVL Trees by Caxton C. Foster (June,1972).

TN/CS/00034 Conditional Syntactic Specification by J. Dorocak and John A.N. Lee (September,1972).

TN/CS/00035 A Formal Definition of Mini Language Number 7,"Dynamic Type Checking" by Edward G. Fisher (October,1972).

TN/CS/00036 Data Administration, Data Independence, and the DBTG Report by Robert W. Taylor (October 1973).