(18.145) MATH. FROM THE CONSTRUCTIVE PT. OF VIEW

How to Compile Mathematics into Algol

By Errett Bishop

Historically, the search for algorithms has been largely ad hoc. Typically, one proves an existence theorem, and afterward looks for an algorithmic version. The business of constructive mathematics is to derive algorithms on a systematic basis. Constructive mathematics is distinguished from numerical analysis in that the numerical analyst looks for practical (or efficient) algorithms. Sometime in the future, when constructive mathematics is better developed, more attention will undoubtedly be paid to questions of efficiency.

It is not premature to look for applications of constructive mathematics to the theory of computation. In this paper we shall investigate the relationship of a certain known formalization Σ of constructive mathematics to certain known programming languages. More precisely, we shall indicate how to compile Σ into algol. The possibility of such a compilation demonstrates the existence of a new type of programming language, one that contains theorems, proofs, quantifications, and implications, in addition to the more conventional facilities for specifying algorithms. The use of a formal mathematical system as a programming language presupposes that the system has a constructive interpretation. Since most formal systems have a classical, or nonconstructive, basis (in particular, they contain the law of the excluded middle), they cannot be used as programming languages.

The role of formalization in constructive mathematics is completely distinct from its role in classical mathematics. Unwilling--indeed unable, because of his education--to let mathematics generate its own meaning, the classical

mathematician looks to formalism, with its emphasis on consistency (either relative, empirical, or absolute), rather than meaning, for philosophical relief. For the constructivist, formalism is not a philosophical out; rather it has a deeper significance, peculiar to the constructivist point of view. Informal constructive mathematics is concerned with the communication of algorithms, with enough precision to be intelligible to the mathematical community at large. Formal constructive mathematics is concerned with the communication of algorithms with enough precision to be intelligible to machines.

Our program is first to present the formal language Σ, and to discuss its constructive interpretation. Rather than show how to compile Σ directly into algol, we show how to compile it into a certain subset Σ_0, whose relation to the conventional programming languages such as algol is much more immediate. Then we discuss the compilation of Σ_0 into algol. The method of compilation is somewhat devious. The reason is that certain algorithms of Σ (those which involve implication) are concerned with converting the proof of one formula into the proof of another. Such algorithms, which do not correspond in a direct way to anything in the conventional programming languages, are intimately intertwined with the more directly numerical algorithms of Σ. Thus we are faced with the problem that if we are going to compile the directly numerical algorithms of Σ, we are going to be forced to somehow compile the proof-conversion algorithms as well.

The present paper is self contained-no previous acquaintance with constructive mathematics or its philosophy is assumed. However, it would be helpful to have a nodding acquaintance with algol,

with constructive mathematics (as in [1] through Chapter 3), and with formal mathematical systems (as in [3] through Chapter 4).

The language Σ is somewhat meagre as a vehicle for constructive mathematics (measure theory, for example), because the only induction it contains is simple induction over the integers. Hopefully the ideas presented here will remain applicable to the compilation of more powerful languages.

The terms of Σ are formed from variables ranging over certain constructively defined sets, and from constants denoting particular elements of those sets. To each of the sets in question we associate a type, which is a string of symbols that denotes the set. The types are formed inductively according to the following rules (1T), (2T), and (3T).

(1T) The symbol $\mathbb{Z}$ is a type, denoting the set of nonnegative integers.

(2T) If T1, ... , Tn are types, denoting respectively the sets S1, ... , Sn, then (T1, ... , Tn) is a type, denoting the cartesian product of those sets.

(3T) If T1 and T2 are types, then $(T1 \rightarrow T2)$ is a type, denoting the set of all (constructively defined) functions from the set denoted by T1 to the set denoted by T2.

An element of the set denoted by T will be called a functional of type T. What symbols or strings of symbols we use for variables and constants does not matter very much, except that to each variable and constant is associated a type, to indicate that the variable ranges over the corresponding set or, in the case of a constant, that it denotes an element of the corresponding set. Only those functionals defined in Σ (a concept to be explained later) will be denoted by constants. The constant "0" denotes the integer 0.

Certain strings of symbols are called terms. The following rules prescribe how terms are inductively formed, and how a type is associated to each term.

(We assume without explicit mention, both in the construction of terms and the construction of formulas, that parentheses are inserted wherever necessary to avoid ambiguity or to achieve clarity, according to the usual conventions.)

(1t) Each constant or variable of type T is a term of type T.

(2t) If t is a term of type (T1 → T2) and t1 is a term of type T1, then (t; t1) is a term of type T2.

(3t) If t is a term of type T ≡ (T1, ... , Tn), then for each integer i with $1 \leq i \leq n$ the expression (t; i) is a term of type Ti.

(4t) If ti is a term of type Ti, for $1 \leq i \leq n$, then (t1, ... , tn) is a term of type (T1, ... , Tn).

(5t) If t is a term of type $\mathbb{Z}$, then t′ is a term of type $\mathbb{Z}$.

(6t) If t is a term of type T, and x1, ... , xn are distinct variables of types T1, ... , Tn respectively, then λx1, x2, ... , xn; t; is a term of type ((T1, ... , Tn) → T).

(7t) If t1, ... , tn are terms of type $\mathbb{Z}$, and u1, ... un are terms of type T, then [t1 : u1; ... ; tn : un] is a term of type T.

(8t) If t1 and t2 are terms of type $\mathbb{Z}$, and t3 is a term of type T, and t4 a term of type (($\mathbb{Z}$, T) → T), then ind(t1, t2, t3, t4) is a term of type T.

A term of type T is to be regarded as representing a functional of type T, in the sense that whenever a functional of the requisite type is substituted for each of the free variables (a concept to be explained later) of t, the result denotes a functional of type T.

The rules for the formation of terms are interpreted as follows. Rule (1t) is self-explanatory. Rule (2t) concerns the evaluation of a functional x at an argument x1. (We use the somewhat unconventional notation (x; x1) for

the result of this evaluation to avoid confusion with another notation to be defined later.) Rule (3t) concerns the selection of the i^{th} element (x; i) of an n-tuple x. Rule (4t) concerns the creation of an n-tuple from its components. Rule (5t) concerns the operation of passing from an integer n to its successor n′. The term λx1, ... , xn; t; of rule (6t) is to be read "the function whose value at (x1, ... , xn) is t(x1, ... , xn)", where we have written t(x1, ... , xn) in place of t in order to draw attention to the free occurences (if any) of the variables x1, ... , xn in t. The term [t1 : u1; ... ; tn : un] of (7t) is to be regarded as representing the same functional as is represented by ui, where i is the least integer ($1 \leq i \leq n$) such that $ti \neq 0$. In case $ti = 0$ for all i, we are not interested in what the term represents, and so may define it to represent the same functional as represented by un. The term ind(t1, t2, t3, t4) of (8t) for given values of the free variables denotes the functional f defined as follows. In case $t2 = t1$, f is the functional denoted by t3. In cast $t2 > t1$, f is functional denoted by the term

$$\text{ind}(t1', t2, (t4; (t1, t3)), t4).$$

We next give inductive rules for the construction of formulas. Like a term, a formula is a string of symbols. The simple formulas of Σ have either the form $t1 = t2$ or the form $t1 \neq t2$, where t1 and t2 are terms of type Z. Since the comparison of two constructively defined integers to see whether they are equal is a finite process, each simple formula A represents a (finitely) decidable statement, in that for given values of the free variables of A it denotes a decidable statement. The following three rules generate all formulas.

(1F) Each simple formula is a formula.

(2F) For each formula A and each variable x, ∃xA and ∀xA are formulas

(sometimes written as $\exists x A(x)$ and $\forall x A(x)$ in order to call attention to the free occurences of x in A).

(3F) If A and B are formulas, so are $A \wedge B$, $A \vee B$, and $A \to B$.

Before proceding, we shall define the concept of a free occurence of a variable x in a formula A, or a term t. In the case of a term t, an occurence of x is free if it is not an occurence of one of the variables $x1, \ldots, xn$ in a subterm of the form $\lambda x1, \ldots, xn; t1;$. For example, the first occurence of the variable bleep in the term ((blip; bleep); λ bleep, v; u + 1;) is free, and the second is not. In the case of a formula A, an occurence of x is free if it satisfies the above requirement and in addition does not occur in a subformula of the form $\forall x A(x)$ or $\exists x A(x)$. For example, the last occurence of x in

$$\exists x(\lambda x; y; = (u;. x)) \vee x = 1$$

is free, and the others are not.

Each formula of Σ represents a constructively meaningful assertion, in that it denotes a constructively meaningful assertion for given values of the free variables, if we interpret $\wedge$, $\vee$, $\to$, $\exists$, and $\forall$ in the constructive (Brouwerian) sense. Here is a brief summary of Brouwer's interpretations. (The interpretations hold for all fixed values of the free variables).

(a) $A \wedge B$ asserts A and also asserts B.

(b) $A \vee B$ either asserts A or asserts B, and we have a finite method for deciding which of the two it does assert.

(c) $A \to B$ asserts that if A is true, then so is B. (To prove $A \to B$, we must give some method that converts each proof of A into a proof of B.)

(d) $\forall x A(x)$ asserts that $A(f)$ holds for each (constructively) defined functional f of the same type as the variable x, where $A(f)$ is obtained

from $A(x)$ by substituting f for all free occurences of x.

(e) $\exists x A(x)$ asserts that we know an algorithm for constructing a functional f for which $A(f)$ holds.

It may be helpful to remark some additional aspects of the functionals f with which we are concerned. In case f has type $(T1, \ldots, Tn)$, to construct f we must construct a functional fi of type Ti, $1 \leq i \leq n$; and f is regarded as <u>equal</u> to a second such functional g if $fi = gi$ for each $i(1 \leq i \leq n)$. In case f has type $(T1 \rightarrow T2)$, to construct f we must give an algorithm that to each functional α of type $T1$ associates a functional $(f; \alpha)$ of type $T2$, such that $(f; \alpha) = (f; \beta)$ whenever $\alpha = \beta$; and we say that $f = g$ if $(f; \alpha) = (g; \alpha)$ for all α of type $T1$. (Thus equality of functionals is defined inductively).

In case a formula $A(x1, \ldots, xn)$ of Σ has the distinct free variables $x1, \ldots, xn$ and no others, it is customary to regard A when standing alone as denoting the same statement as the formula $\forall x1 \ldots \forall xn\, A(x1, \ldots, xn)$. This is called the <u>generality interpretation</u>. According to this interpretation the formula $x = x$ is true whereas $x = y$ is false.

To complete the formulation of the language Σ, we must give formal methods for deriving formulas that are (constructively) true (according to the generality interpretation). This is done in two steps. First, certain formulas are distinguished as axioms. These formulas must be true. Second, certain rules of inference, each of which produces a formula (called the <u>conclusion</u>) from certain other formulas (called the <u>premises</u>) are given. These rules must always produce true conclusions from true premises. The reader should check that each of the axioms and rules given below satisfies these requirements. It follows that each theorem of Σ, that is each formula derived from the axioms by successive

applications of the rules to formulas already derived, is true. Each of the rules will have either one premise or two. An axiom could be regarded as a rule with 0 premises, but this is not the customary point of view. In the statement of the axioms and rules we use the letters A, B, and C to denote arbitrary formulas, the letters x, y, and z to denote arbitrary variables, and so forth. Thus each axiom or rule as stated in fact develops into an infinite number of axioms or rules, obtained by replacing the symbols A, B, C by particular formulas, the symbols x, y, z by particular variables, and so forth. The first group of axioms and rules concerns the propositional calculus. If we were to add the axiom of the excluded middle $(A \vee (A \to 0 = 1))$ to this group, the resulting extension of Σ would be a formalization of classical mathematics, of comparable generality.

(1A) $A \to A$.

(1R) If A, then $B \to A$.

(2R) If A and $A \to B$, then B.

(3R) If $A \to B$ and $B \to C$, then $A \to C$.

(2A) $A \wedge B \to A$, $B \wedge A \to A$, $A \to A \vee B$, $B \to A \vee B$.

(4R) If $A \to C$ and $B \to C$, then $A \vee B \to C$.

(5R) If $C \to A$ and $C \to B$, then $C \to A \wedge B$.

(6R) If $A \to (B \to C)$, then $(A \wedge B) \to C$.

(7R) If $(A \wedge B) \to C$, then $A \to (B \to C)$.

(3A) $0 = 0' \to A$.

The second group concerns the quantifiers $\exists$ and $\forall$, and together with the first group axiomitizes the constructive predicate claculus. The letter x stands for an arbitrary variable, and the letter t an arbitrary term of the same type as the variable x. As before, we use the notation $A(x)$ to

draw attention to the free occurences of x in A, and $A(t)$ to denote the result of substituting the term t for each of those free occurences.

(8R) If $A \to B(x)$, then $A \to \forall x B(x)$.

(9R) If $B(x) \to A$, then $\exists x B(x) \to A$.

(4A) $\forall x A(x) \to A(t)$.

(5A) $A(t) \to \exists x A(x)$.

In (8R) and (9R), the variable x is assumed not to occur free in A. In (4A) and (5A), the term t is assumed to be free for x in $A(x)$, which means that no free occurence of x in $A(x)$ lies in the scope of a quantifier $\forall y$ or $\exists y$, where y is a variable that is free in t. For example, the term y is not free for x in the formula $A(x) \equiv \forall y(x = y)$, because x lies in the scope of $\forall y$. This is good, because otherwise (5A) would give the axiom $\forall y(y=y) \to \exists x \forall y(x=y)$.

The next axiom, for obvious reasons, is called the axiom of choice. The symbols x, y, and z stand for arbitrary variables, with x of type T1, y of type T2, and z of type $(T1 \to T2)$. It is assumed that z does not occur free in $A(x, y)$.

(6A) $\forall x \exists y A(x, y) \to \exists z \forall x A(x, (z; x))$.

The constructive truth of (6A) is open to question. To see this, let us assume $\forall x \exists y(x, y)$ and try to show $\exists z \forall x A(x, (z; x))$. Then for each functional f of type T1 we have a proof of $\exists y A(f, y)$, which means we have a construction of a functional $\tilde{f}$ of type T2 such that $A(f, \tilde{f})$. It is natural to try to define a functional h of type $(T1 \to T2)$ by setting $(h; f) \equiv \tilde{f}$ for each f of type T1. If this definition is valid, then we have $A(f, (h; f))$, and therefore $\exists z \forall x A(x, (z; x))$. The difficulty is that the operation h could conceivably fail to be a functional-there might exist equal functionals f_1 and f_2 of type T1 with $(h; f1) \neq (h; f2)$. In case x has type $T1 \equiv \mathbb{Z}$, this difficulty

can be resolved as follows. For each integer n, let $n0$ be a canonical description of n (for instance, n expressed in decimal notation). Set $(h0; n) \equiv (h; n0)$. Then $h0$ satisfies $A(n, (h0; n))$ for all n, and $h0$ is a functional. Thus (6A) is valid in case $T1 \equiv \mathbf{Z}$. This trick does not work for higher types, because no canonical representation exists. However, it is intuitively very plausible that for sets of the types being considered every constructively definable operation is in fact a function. Whether we want to accept (6A) as constructively true for higher types seems to be a matter of personal preference. Both empirical and formal evidence indicates that the loss of deductive power in restricting (6A) to the case $T1 \equiv \mathbf{Z}$ would not be significant. On the other hand, the fact that the formal system, including (6A) for higher types, can be compiled is evidence in favor of accepting the general case.

In order to formulate the next few axioms, a definition is in order. Let $t1$ and $t2$ be terms of Σ of the same type T. In case $T \equiv \mathbf{Z}$, the expression $t1 = t2$ is a simple formula. In the general case, we shall associate a formula of Σ to the expression $t1 = t2$, and use the expression as an abbreviation of the associated formula. The definition is by induction, corresponding to the inductive definition (1T) - (3T) of type. In case $T \equiv (T1, \ldots, Tn)$, we take $t1 = t2$ to be an abbreviation of

$$(t1; 1) = (t2; 1) \wedge \ldots \wedge (t1; n) = (t2; n).$$

In case $T \equiv (T1 \to T2)$, we take it to be an abbreviation of

$$\forall x[(t1; x) = (t2; x)],$$

(where x is any variable of type $T1$ not occuring free in either $t1$ or $t2$).

The next axiom states that any assertion of our language that holds for a given functional holds for any equal functional. The (informal) proof is by

induction on the structure of the formula A, of course.

(7A) $(t1 = t2 \wedge A(t1)) \rightarrow A(t2)$.

The next axiom, in which m and n are integers with $1 \leq m \leq n$, expresses the meaning of (3t).

(8A) $((t1, \ldots, tn); m) = tm$

The next axioms, under appropriate assumptions as to types, express the meanings of (5t), (6t), and (7t) respectively.

(9A) $(\lambda x1, \ldots, xn; t(x1, \ldots, xn);; (t1, \ldots, tn)) = t(t1, \ldots, tn)$

(10A) $u1 = 0 \wedge \ldots \wedge u(i - 1) = 0 \wedge ui \neq 0 \rightarrow [u1 : t1; \ldots; un; tn] = ti$

for $1 \leq i \leq n$

(11A) $t1 = t2 \rightarrow ind(t1, t2, t3, t4) = t3$

$t1 \neq t2 \rightarrow ind(t1, t2, t3, t4) = ind(t1', t2, (t4; (t1, t3)), t4)$

The next rule is mathematical induction. The variable x has type Z.

(10R) If A(0) and $A(x) \rightarrow A(x')$, then A(x).

The final group of axioms pertains to equality and inequality. The variables and terms all have type Z.

(12A) $x = x$, $(x = y \wedge z = y) \rightarrow x = z$

(13A) $x = y \rightarrow t(x) = t(y)$

(14A) $x' = 0 \rightarrow 0' = 0$

(15A) $x' = y' \rightarrow x = y$, $x = y \rightarrow x' = y'$

(16A) $x = y \vee x \neq y$, $(x = y \wedge x \neq y) \rightarrow 0 = 1$

The final rule concerns the introduction of constants.

(11R) If $\exists x A(x)$, then $A((c0; (x1, \ldots, xn)))$

where $x1, \ldots, xn$ are the distinct free variables of $\exists x A(x)$, written say in order of first occurence, and c0 is any available constant of the appropriate type. The interpretation is that c0 denotes the functional $\alpha(P)$, constructed according to the proof P of $\exists x A(x)$, satisfying

$A((\alpha(P); x1, \ldots, xn))$. The functional $\alpha(P)$ is said to be defined in Σ. Only functionals defined in Σ are denoted by constants.

This completes the definition of the language Σ. (A language for classical mathematics, of comparable power, could be obtained from Σ by adding the single axiom (of the excluded middle) $A \vee (A \to 0 = 0')$.) According to the interpretation of the formulas, each functional $\alpha(P)$ defined in Σ is constructively defined. In particular, if $\alpha(P)$ is an integer, we should be able to program a computer to compute it. Even more: The program for computing $\alpha(P)$ should be obtainable from the proof P by some mechanical process. In other words, we should be able to write, in any sufficiently powerful string-manipulation programming language, a routine whose argument is a (suitably encoded) proof P in Σ and whose value, in case $\alpha(P)$ has type $\mathbb{Z}$, is an algol procedure for the computation of the corresponding integer $\alpha(P)$. The construction of such a routine (in technical jargon, a compiler from Σ to algol) would of course involve many man-hours, or perhaps even man-years, of work. This does not concern us here-we are only interested in the general method of building the compiler. Presumably it will operate somewhat as follows. Each axiom of Σ will be translated into an algol procedure. Each rule of inference will be translated into an algol procedure, whose parameters include the algol procedures which translate the proofs of the premises. Since each step of each proof in Σ involves either an axiom or a rule of inference, each proof P in Σ will thereby be translated into an algol procedure $p(P)$, in such a way that if $\alpha(P)$ has type $\mathbb{Z}$ then one of the effects of $p(P)$ will be to assign to a certain algol variable the value $\alpha(P)$. The construction of such a compiler is not quite straightforward, because proofs in Σ describe not only algorithms for constructing functionals of the various types, but also algorithms for converting proofs into proofs, and mixtures of the two.

Techniques for interpreting arbitrary proofs in Σ as algorithms for the construction of functionals are found in [2]. In the present context, these techniques can be intuitively expressed as follows. To each proof P in Σ we associate a constant-free term $\{t : P\}$ of Σ which encodes certain information $I(P)$ about P. The information in question depends on the structure of the formula A being proved, and is defined inductively as follows. In case A is existence-free, which means it contains no existential quantifiers $\exists$ or disjunctions $\vee$, no information is encoded. In case $A \equiv A(x1, \ldots, xn)$ has the free variables $x1, \ldots, xn$, the information $I(P)$ encompasses the totality of the information $I(P')$, where A' denotes the assertion obtained by substituting arbitiary functionals of the appropriate types for the variables $x1, \ldots, xn$ and P' denotes the (informal) proof of A' derived from the proof P of A. In the remaining cases we assume A has no free variables. In case $A \equiv A1 \wedge A2$, the information $I(P)$ encompasses both $I(P1)$ and $I(P2)$, where $P1$ and $P2$ are the corresponding proofs of $A1$ are $A2$ respectively. In case $A \equiv A1 \vee A2$, the information $I(P)$ tells which Ai is being proved, and encompasses the information $I(Pi)$ for the proof Pi of that Ai. In case $A \equiv A1 \rightarrow A2$, then $I(P)$ tells us how to translate the information $I(P1)$ for an arbitrary proof $P1$ of $A1$ into the information $I(P2)$ for the corresponding proof $P2$ of $A2$. In case $A \equiv \exists x A1(x)$, then $I(P)$ gives the algorithm for the construction of the functional $\alpha(P)$, and also gives the information $I(P')$ for the corresponding proof P' of $A1(\alpha(P))$. In case $A \equiv \forall x A(x)$, then $I(P)$ is the same as $I(P')$, where P' is the corresponding proof of $A(x)$.

To realize these ideas, we define a generalized formula (or g-formula) A to be a symbol-string obtained from a formula $A0$ of Σ by substituting generalized constants of the requisite types for certain of the free variables of Σ. A generalized constant of a certain type is a symbol string denoting a particular

functional of that type. For example, if A0 is the formula (y; x) = 0, where the variables x and y have types $\mathbb{Z}$ and $(\mathbb{Z} \to \mathbb{Z})$ respectively, then if the generalized constant "primefac" is chosen to denote the functional of type $(\mathbb{Z} \to \mathbb{Z})$ whose value at the integer 0 is 0 and at any positive integer n is the number of distinct prime factors of n, then (primefac; x) = 0 is a generalized formula A(x) (which is true when x has the value 0 or 1, is false for all other values of x, and is false according to the generality interpretation). We define a <u>generalized term</u> (or g-term) to be a symbol-string t obtained from a term t0 of Σ by substituting generalized constants of the appropriate types for certain of the free variables of t0. To each g-formula A we associate a type T(A). In general T(A) is defined to be the same as T(A0), where A0 is the formula from which A was obtained. For a formula A, the inductive definition goes as follows.

(a) In case A is existence free, $T(A) \equiv \mathbb{Z}$.

For the remaining cases, A is assumed to be <u>existential</u>-to contain either a $\exists$ or a $\vee$.

(b) In case $A \equiv A1 \wedge A2$, then $T(A) \equiv (T(A1), T(A2))$.

(c) In case $A \equiv A1 \vee A2$, then $T(A) \equiv (\mathbb{Z}, T(A1), T(A2))$.

(d) In case $A \equiv A1 \to A2$, then $T(A) \equiv (T(A1) \to T(A2))$

(e) In case $A \equiv \exists x A1(x)$, then $T(A) \equiv (T(x), T(A1(x)))$, where T(x) is the type of x.

(f) In case $A \equiv \forall x A1(x)$, then $T(A) \equiv (T(x) \to T(A1(x)))$.

We next give an inductive definition of what it means for a generalized term t of type T(A) to <u>compile</u> a constructive proof P, formal or informal, of a generalized formula A.

(a) In case A has free variables, then t compiles P if for each assignment a of a functional of the requisite type to each of the free variables of

A, t_a compiles the corresponding proof P_a of A_a, where t_a is the generalized term obtained from t by replacing each free variable by the corresponding functional, and A_a is the generalized formula obtained from A by the same process. For the remaining cases, we may assume that A has no free variables.

(b) In case $A \equiv A1 \wedge A2$, then t compiles P if (t; 1) compiles the corresponding proof of A1 and (t; 2) compiles the corresponding proof of A2.

(c) In case $A \equiv A1 \vee A2$, then t compiles P if (t; 1) = 0 in case A1 is being proved and (t; 1) $\neq$ 0 is case A_2 is being proved, and if (t; i + 1) compiles the corresponding proof of Ai in case Ai is being proved.

(d) In case $A \equiv A1 \rightarrow A2$, then t compiles P if whenever t1 compiles a proof P1 of A1, the term (t; t1) compiles the corresponding proof P2 of A2.

(e) In case $A \equiv \exists x A1(x)$, then t compiles P if (t; 1) = α(P) (for all values of the free variables of (t; 1), if any) and (t; 2) compiles the corresponding proof of A1(α(P)).

(f) In case $A \equiv \forall x A1(x)$, then t compiles P if (t; x) compiles the corresponding proof of A1(x).

Our fundamental result is the existence of an algorithm AL, realizable in any general string-manipulation programming language, which to each proof P in Σ associates a constant-free term {t : P} of Σ that compiles P. The algorithm AL will be constructed in terms of three other algorithms, AL1, AL2, and AL3.

The algorithm AL1 associates to each axiom A of Σ a term t of Σ that compiles the proof P of A and contains only constants that occur in A.

The algorithm AL2 associates to each rule of inference of Σ, having a single permise B and conclusion A, a constant-free term $t(\beta)$ of Σ, containing the distinguished free variable β, such that if a is any assignment of all variables

occuring free in either A or B, exclusive of the variable denoted by x in rules (8R) and (9R), then for any g-term u compiling a proof Q_a of B_a, the g-term $t_a(u)$ compiles the corresponding proof P_a of A_a (Here of course β is assumed not to occur free in either of the formulas A or B, so that $t_a(\beta)$ denotes the term obtained by making the substitutions a in $t(\beta)$, and $t_a(u)$ the term obtained by substituting u for β in $t_a(\beta)$.)

The algorithm AL3 associates to each rule of inference of Σ, having two premises B and C and conclusion A, a term $t(\beta, \lambda)$ of Σ, containing the distinguished free variables β and λ, such that if a is any assignment of all variables occuring free in the formulas A, B, or C, with the exception of the variable denoted by x in rule (10R), then for any g-terms u and v compiling proofs Q_a of B_a and R_a of C_a respectively, the g-term $t_a(u, v)$ compiles the corresponding proof P_a of A_a.

Before giving AL1, AL2, and AL3, we show how they can be used to obtain AL. To each proof P in Σ we wish to associate a term $\{t : P\}$ as described. This is done by induction on the proof-tree of P. In case P is the assertion that the associated formula A is an axiom, then AL1 produces a term t that compiles P, but t may contain certain of the constants ci occuring in A. Now the instances of (11R) which define those constants are to be regarded as part of the proof-tree of P. Thus AL is to be regarded as already defined for the proofs Pi of the formulas $\exists xi Ai(xi)$ that go into defining the constants ci. Thus a term $ti \equiv (\{t : Pi\}; 1)$ equal to ci is already available. We obtain $\{t; P\}$ by substituting ti for each occurence of ci in t.

Next consider the case in which the proof P of A consists in deducing A as the conclusion of a rule of inference having a single premise B with

proof Q. Let $t(\beta)$ be the corresponding term constructed by AL2. By the inductive hypothesis, $\{t : Q\}$ is already defined. Write $\{t : P\} \equiv t(\{t : Q\})$. To show that $\{t : P\}$ compiles P, it is enough to show that $\{t : P\}_a = t_a(\{t : Q\}_a)$ compiles P_a for each assignment a of the free variables of A and B (with the exception noted above for rules (8R) and (9R)). By definition, $\{t : Q\}_a$ compiles the proof Q_a of B_a. Hence $t_a(\{t : Q\}_a)$ compiles P_a.

In the final case, the proof P of A consists in deducing A as the conclusion of a rule of inference having premises B with proof Q and C with proof R. Let $t(\beta, \lambda)$ be the corresponding term constructed by AL3. By the inductive hypothesis, $\{t : Q\}$ and $\{t : R\}$ are already defined. Write $\{t : P\} \equiv t(\{t : Q\}, \{t : R\})$. It is enough to show that $\{t : P\}_a = t_a(\{t : Q\}_a, \{t : R\}_a)$ compiles P_a for an arbitrary assignment a of the free variables of A, B, C (with the exception of x for rule (10R)). The generalized term $u \equiv \{t : Q\}_a$ compiles B_a, and $v \equiv \{t : Q\}_a$ compiles C_a. Hence $t_a(u, v)$ compiles P_a.

It only remains to consider each of the axioms and rules in turn, and define the appropriate term in each case. The following notation will be used throughout. We shall use A to denote the axiom being considered or, in the case of a rule, the conclusion. The premise of a single premise rule will be denoted by B, and the premises of a double premise rule by B and C. An arbitrary assignment of the free variables of A (in the case of an axiom), or A, B (in the case of a single premise rule) or A, B, C (in the case of a double premise rule), with the exceptions noted above, will be denoted by a. The subscript a on a term or formula will denote the generalized term or generalized formula obtained by making the substitutions of the functionals assigned by a for the corresponding free variables of the given term or formula.

An arbitrary proof of B_a will be denoted by Q_a and an arbitrary proof of C_a by R_a. An arbitrary generalized term compiling Q_a will be denoted by u and an arbitrary generalized term compiling R_a by v. The intended proof of A_a (obtained from the intended proof P of A in case A is an axiom, from the given proof Q_a of B_a in the case of a rule with one premise, and from the given proofs Q_a of B_a and R_a of C_a in the case of a rule with two premises) will be denoted by P_a.

(1A) In this case, $A \equiv A' \to A'$ is an axiom. We set $t \equiv \lambda x;\ x;$, where x is a variable of type $T(A')$. We must show that if the g-term t' compiles a proof P'_a of A'_a, then $(t;\ t') = t'$ compiles the corresponding proof P''_a of A'_a. This follows from the fact that P''_a is the same proof of A'_a as P'_a (we had this in mind when we asserted $A' \to A'$ to be an axiom).

(1R) In this case, the premise is B, and the conclusion is $A \equiv B' \to B$. We set $t(\beta) \equiv \lambda x;\ \beta;$, where x is any variable of type $T(B')$. To show that $t_a(u) = \lambda x;\ u;$ compiles A_a, we must show that if Q'_a is any proof of B'_a and u' any g-term that compiles Q'_a, then $(t_a(u);\ u') = u$ compiles the corresponding proof Q''_a of B_a. This follows from the fact that Q''_a is the same proof of B_a as Q_a.

(2R) In this case the premises are B and $C \equiv B \to A$, and the conclusion is A. We set $t(\beta, \gamma) \equiv (\gamma;\ \beta)$. We must show that $(v;\ u)$ compiles the proof P_a of A_a. Since u compiles the proof Q_a of B_a and v compiles the proof R_a of $C_a \equiv B_a \to A_a$, the g-term $(v;\ u)$ compiles the corresponding proof of A_a, which is just P_a.

(3R) In this case, $B \equiv B' \to D$, $C \equiv D \to C'$, and $A \equiv B' \to C'$. We set $t(\beta, \gamma) \equiv \lambda x;\ (\gamma;\ (\beta;\ x));$. To show that $t(u, v)$ complies P_a, consider any g-term u' that compiles a proof Q'_a of B'_a. We must show that $(t(u, v);\ u') = (v;\ (u;\ u'))$ compiles the corresponding proof R'_a of C'_a.

Now $(u;\ u')$ compiles the proof S_a of D_a obtained from the proof Q'_a of B'_a and the proof B_a of $B_a \equiv B'_a \to D_a$. Hence $(v;\ (u;\ u'))$ compiles the proof of C'_a obtained from the proof S_a of D_a and the proof R_a of $C_a \equiv D_a \to C'_a$; and that proof is the same as R'_a.

(2A) We consider only the first and third cases of (2A). In case 1, the axiom A is $A' \wedge A'' \to A'$. We set $t \equiv \lambda x;\ (x;\ 1);$, where x is any variable of type $T(A' \wedge A'')$. To show t compiles A, we must show that if s compiles any proof L of $A'_a \wedge A''_a$ then $(t_a;\ s) = (s;\ 1)$ compiles the corresponding proof of A'_a. This is true by definition. In the third case of (2A), the axiom A is $A' \to A' \vee A''$. We set $t \equiv \lambda x;\ (0,\ x,\ y);$, where x is any variable of type $T(A')$ and y is any variable of type $T(A'')$. To show t compiles A, we must show that if t' compiles any proof P'_a of A'_a then $(t_a;\ t') = (0,\ t',\ y)$ compiles the corresponding proof S_a of $A'_a \vee A''_a$. Now $((0,\ t',\ y);\ 1) = 0$ means that S_a must actually choose A'_a to prove, which it does. We must check also that the corresponding proof of A'_a, which is P'_a, is compiled by $((0,\ t',\ y);\ 2) = t'$, which it is.

(4R) In this case $B \equiv B' \to D$, $C \equiv C' \to D$, and $A \equiv B' \vee C' \to D$. We set $t(\beta,\ \gamma) \equiv \lambda x;\ [(x;\ 1) : (\gamma;\ (x;\ 3));\ 1 : (\beta;\ (x;\ 2))];$. To show that $t_a(u,\ v)$ compiles A_a, consider a g-term s_a that compiles a proof L_a of $B'_a \vee C'_a$. We must show that $(t_a(u,\ v);\ s_a)$ compiles the corresponding proof S_a of D_a. There are two cases to consider, depending on whether L_a selects B'_a or C'_a to prove. We consider only the first case, the argument in the second case being similar. Let Q'_a be the proof of B'_a provided by L'_a. The proof S_a of D_a is the same as the proof of D_a provided by the proof Q'_a of B'_a and the proof Q_a of $B_a \equiv B'_a \to D_a$.

Since $(s_a; 2)$ compiles Q'_a and u compiles Q_a, it follows that $(u; (s_a; 2))$ compiles S_a. On the other hand, since $(s_a; 1) = 0$, we have $(t_a(u, v); s_a) = (u; (s_a; 2))$.

(5R) In this case, $B \equiv D \to B'$, $C \equiv D \to C'$, and $A \equiv D \to B' \wedge C'$. Set $t(\beta, \gamma) \equiv \lambda x; ((\beta; x), (\gamma; x));$. To show $t_a(u, v)$ compiles A_a, consider a g-term w compiling a proof S'_a of D_a. We must show that $(t_a(u, v); w) = ((u; w), (v; w))$ compiles the corresponding proof of $B'_a \wedge C'_a$, which means that $(u; w)$ compiles the corresponding proof of B'_a and $(v; w)$ the corresponding proof of C'_a. Now the proofs in question are respectively the same as the proof Q'_a of B'_a obtained from the proof S'_a of D_a and the proof Q_a of B_a, and the proof R'_a of C'_a obtained from the proof S'_a of D_a and the proof R_a of C_a. Since w compiles S'_a and u compiles Q_a, it follows that $(u; w)$ compiles Q'_a. Similarly, $(v; w)$ compiles R'_a.

(6R) In this case, $B \equiv D \to (E \to F)$ and $A \equiv (D \wedge E) \to F$. We set $t(\beta) \equiv \lambda x; ((\beta; (x; 1)); (x; 2));$ where x is any variable of type $T(D \wedge E)$. To show that $t_a(u)$ compiles P_a, consider any proof R_a of $D_a \wedge E_a$, and any g-term w that compiles R_a. We must show that $(t_a(u); w)$ compiles the corresponding proof S_a of F_a. Now $(w; 1)$ and $(w; 2)$ respectively compile the proofs R'_a of D_a and R''_a of E_a provided by R_a. Hence $(u; (w; 1))$ compiles the proof S'_a of $E_a \to F_a$ provided by R'_a and Q_a, and $((u; (w; 1)); (w; 2))$ compiles the proof of F_a provided by R''_a and S'_a, which is the same proof as S_a. It only remains to note that $(t_a(u); w) = ((u; (w; 1)); (w; 2))$.

(7R) In this case, $B \equiv (D \wedge E) \to F$ and $A \equiv D \to (E \to F)$. We set $t(\beta) \equiv \lambda x; \lambda y; (\beta; (x, y));;$. To show $t_a(u)$ compiles A_a, consider a

g-term w1 compiling a proof R'_a of D_a. We must show that $(t_a(u); w1) = \lambda y; (u; (w1, y))$; compiles the corresponding proof S'_a of $E_a \to F_a$, which means that $(u; (w1, w2))$ compiles the corresponding proof S_a of F_a whenever w2 compiles a proof R''_a of E_a. Now $(w1, w2)$ compiles the proof R_a of $E_a \wedge F_a$ obtained from the proofs R'_a and R''_a. Hence $(u; (w1, w2))$ compiles the proof of F_a obtained from R_a and Q_a, which is the same as S_a.

(3A) In this case, $A \equiv 0 = 0' \to A'$. We set $t \equiv \lambda x; y$; where x is any variable of type $\mathbb{Z}$ and y is any variable of type $T(A')$. We must show that if R_a is any proof of $0 = 0'$, and u compiles R_a, then $(t; u) = y$ compiles the associated proof of A'. Since there are no proofs of $0 = 0'$, this is clear.

(8R) In this case $B \equiv E \to F(x)$ and $A \equiv E \to \forall x F(x)$. We set $t(\beta) \equiv \lambda y; \lambda x; (\beta; y)$ where y is any variable of type $T(E)$. If w is any g-term that compiles a proof R_a of E_a, we must show that $(t_a(u); w) = \lambda x; (u(x); w)$; compiles the corresponding proof S_a of $\forall x F_a(x)$. For this, it is enough to show that for each functional f of type $T(x)$, the term $(u(f); w)$ compiles the corresponding proof S'_a of $F_a(f)$. Since $u(x)$ compiles the proof Q_a of $E_a \to F_a(x)$, the term $u(f)$ compiles the corresponding proof Q'_a of $E_a \to F_a(f)$. Hence $(u(f); w)$ compiles the proof of $F_a(f)$ obtained from the proofs R_a of E_a and Q'_a of $E_a \to F_a(f)$, which is the same as the proof S'_a.

(9R) In this case, $B \equiv E(x) \to F$ and $A \equiv \exists x E(x) \to F$. We set $t(\beta) \equiv \lambda y; ((\lambda x; \beta;; (y; 1)); (y; 2))$; where y has type $T(\exists x E(x))$. If R_a is any proof of $\exists x E_a(x)$, compiled by the g-term w, we must show that $(t_a(u(x)); w) = (u((w; 1)); (w; 2))$ compiles the corresponding proof S_a

of F_a. Now $u((w; 1))$ compiles the proof Q'_a of $B'_a \equiv E_a((w; 1)) \to F_a$. Since $(w; 1)$ is equal to the functional $\alpha(R_a)$, the term $u((w; 1))$ also compiles the proof Q''_a of $E_a(\alpha(R_a)) \to F_a$. Since $(w; 2)$ compiles the proof R'_a of $E_a(\alpha(R_a))$, it follows that $(u((w; 1)); (w; 2))$ compiles the proof of F_a obtained from the proofs R'_a of $E_a(\alpha(R_a))$ and Q''_a of $E_a(\alpha(R_a)) \to F_a$, which is the same proof as S_a.

(4A) In this case, $A \equiv \forall x A'(x) \to A'(w)$, where w is any term free for x in $A'(x)$. We set $t \equiv \lambda y; (y; w)$; where y has type $T(\forall x A'(x))$. We must show that if t' compiles the proof P'_a of $\forall x A'(x)$, then $(t_a; t') = (t'; w$ compiles the corresponding proof P''_a of $A'(w_a)$. This is true, because by defintion $(t'; f)$ compiles the corresponding proof of $A'(f)$ for all functionals f of type $T(x)$.

(5A) In this case, $A \equiv A'(w) \to \exists x A'(x)$ where w is free for x in $A'(x)$. We set $t \equiv \lambda y; (w, y)$; where y has type $T(A'(x))$. Let t' compile the proof P'_a of $A'_a(w_a)$. We must show that $(t_a; t') = (w_a, t')$ compiles the associated proof P''_a of $\exists x A'_a(x)$. Since $\alpha(P''_a) = w_a$, we must first of all have $((w_a, t'); 1) = w_a$, which we do. Second, t' must compile the associated proof of $A'_a(w_a)$, which it does because that proof is the same as P'_a.

(6A) In this case $A \equiv \forall x \exists y A'(x, y) \to \exists z \forall x A'(x, (z; x))$. We set $t \equiv \lambda z; (\lambda x; ((z; x); 1);, \lambda x; ((z; x); 2);); .$ To show that t_a compiles P_a, consider a term t' compiling a proof P'_a of $\forall x \exists y A'_a(x, y)$. We must show that $(t_a; t')$ compiles the corresponding proof P''_a of $\exists z \forall x A'(x, (z;$ This means first, to show that $((t_a; t'); 1) = \lambda x; (t'; x); 1);$ equals $\alpha(P''_a)$ and second, to show that $((t_a; t'); 2) = \lambda x; ((t'; x); 2);$ compiles

the corresponding proof of $\forall x A'_a(x, (\alpha(P''_a); x))$. As to the first statement, it is equivalent to the statement that $((t'; f); 1) = (\alpha(P''_a); f)$ for all functionals f of type $T(x)$. By definition, $((t'; f); 1)$ equals $\alpha(P'_a(f))$, where $P'_a(f)$ is the corresponding proof of $\exists y A'_a(f, y)$. Since $(\alpha(P''_a); f) = \alpha(P'_a(f))$, the first statement holds. As to the second statement, it means that $((t'; f); 2)$ compiles the corresponding proof S_a of $A'_a(f, (\alpha(P''_a); f))$. By definition, $((t'; f); 2)$ compiles the proof of $A'_a(f, \alpha(P'_a(f)))$ obtained from the proof $P'_a(f)$, and therefore compiles the associated proof of $A'_a(f, (\alpha(P''_a); f))$, which is the same as the proof S_a.

(7A) In this case, A'' is the axiom $t_1 = t_2 \to (A'(t_1) \to A'(t_2))$. We set $t \equiv \lambda x; \lambda y; y;;$ where x has type $\mathbb{N}$ and y type $T(A'(t_1))$. If t' compiles a proof P'_a of $t_{1a} = t_{2a}$, we must show that $(t_a; t') = \lambda y; y;$ compiles the corresponding proof P''_a of $A'_a(t_{1a}) \to A'_a(t_{2a})$, which means that if t'' compiles a proof S_a of $A'_a(t_{1a})$ then t'' also compiles the corresponding proof S'_a of $A'_a(t_{2a})$. This follows from the equality $t_{1a} = t_{2a}$.

The only remaining axioms and rules involving existential formulas are (10R) and (11R). For the others, the algorithm can be defined arbitrarily, as long as the term has the right type.

(10R) In this case $B \equiv A(0)$, $C \equiv A(x) \to A(x')$, and $A \equiv A(x)$. We set $t(\beta, \gamma) \equiv \mathrm{ind}(0, x, \beta, \lambda x, y; (\gamma; y);)$, where y has type $T(A(x))$. We must show that the g-term $w(n) \equiv \mathrm{ind}(0, n, u, \lambda n, y; (v(n); y);)$ compiles the proof $P_a(n)$ of $A_a(n)$ for each nonnegative integer n. The proof is by induction on n. When $n = 0$, we have $w(0) = u$, and the proof of $A_a(0)$ is B_a. Thus the assertion is true in this case.

For the inductive step, assume $w(n)$ compiles $P_a(n)$. Since $v(n)$ compiles the proof $R_a(n)$ of $C_a(n) \equiv A_a(n) \to A_a(n')$, it follows that $(v(n); w(n)) = w(n')$ compiles the proof of $A_a(n')$ obtained from $P_a(n)$ and $R_a(n)$, which is the same as $P_a(n')$.

(11R) In this case, $B \equiv \exists x B'(x)$ and $A \equiv B'((c0; x1, \ldots, xn))$. We set $t(\beta) \equiv (\beta; 2)$. We must show that $(u; 2)$ compiles the proof P_a of $(B'((c0; x1, \ldots, xn))_a$. By assumption, $(u; 2)$ compiles the proof Q'_a of $B'_a(c(Q_a))$. Since $c(Q_a) = (c0; x1, \ldots, xn)_a$, we get the desired result.

This completes the definition of AL1, AL2, and AL3. Several comments are in order. First, we have used the fact that if the g-terms t1 and t2 are equal for all values of the free variables, and if t1 compiles a certain proof P, then t2 also compiles P. The proof, of course, is by induction, and involves the consideration of each of the cases (a) - (f) in the definition of compilation in turn. Second, we have spoken very informally of two proofs P and P′ of the same generalized formula A as being the same, and we have assumed that if P and P′ are the same then any g-term that compiles P also compiles P′. It is possible to make the concept of sameness precise. The definition when the proofs P and P′ of the same g-formula A are the same is by induction on the structure of A. In case A has free variables x1, ... , xn, then P and P′ are the same if the corresponding proofs of A(f1, ... , fn) are the same, for all functio f1, ... , fn of the requisite types. For the remaining cases, A is assumed to have no free variables. If $A \equiv A1 \wedge A2$, then P and P′ are the same if the corresponding proofs of Ai are the same (for i = 1 and i = 2). If $A \equiv A1 \vee A$ then P and P′ are the same if the same Ai is being proved, and if the proofs of that Ai are the same. If $A \equiv A1 \to A2$, then P and P′ are the same if for an arbitrary proof of A1, the corresponding proofs of A2 are the same. If

$A \equiv \forall x A1(x)$, the corresponding proofs of $A1(x)$ must be the same. Finally, if $A = \exists x A1(x)$, then P and P' are the same if $\alpha(P) = \alpha(P')$ and if the corresponding proofs of $A1(\alpha(P))$ are the same. An inductive arugment shows that if $P1$ and $P2$ are the same, then any term that compiles $P1$ also compiles $P2$. Third, we have used the fact that if $A(x)$ is a g-formula having a free variable x, if $f1$ and $f2$ are functionals of the same type as x, with $f1 = f2$, and if the g-term t compiles a proof $P1$ of $A(f1)$, then t also compiles the corresponding proof $P2$ of $A(f2)$. This is proved by induction, using the notion of sameness.

Having compiled Σ into the set Σ_0 of constant-free terms of Σ, our next task is to compile Σ_0 into algol. It would be trivial to compile Σ_0 into the latest version algol 68 of algol, since each type in Σ has a corresponding type (called a mode) in algol 68. (See [5] for details.) However, in algol 60, the version for which implementations are presently available, the type structure is much less rich. It is necessary to compile each term of Σ_0 as an integral procedure.

To present the construction, a definition is needed. A type T is <u>prime</u> if it can be obtained inductively from the following rules.

(1T′) $\mathbb{Z}$ is a prime type.

(2T′) If $T1, \ldots, Tn$ are prime types, then so is $(T1, \ldots, Tn)$.

(3T′) If T is a prime type, then so is $(T \to \mathbb{Z})$.

The key remark is that to each type T corresponds a prime type $\sigma(T)$ such that the sets denoted by T and $\sigma(T)$ are in canonical one-one correspondence. For example, if $T \equiv (\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z}))$, then $\sigma(T) \equiv ((\mathbb{Z}, \mathbb{Z}) \to \mathbb{Z})$. This remark makes it straightforward, albeit tedious, to compile Σ_0 into algol 60. We content ourselves with an example. Let t be the term

$$t \equiv \lambda x, y, z, w; \text{ind}(x, y, z, w);$$

where x and y are variables of type $\mathbb{Z}$, where z is a variable of type T, with $\sigma(T) \equiv ((T1, \ldots, Tn) \to \mathbb{Z})$, and w is a variable of type $((T, \mathbb{Z}) \to T)$. Then t is compiled as an integer procedure IND with arguments X, Y, Z1, W1, U1, ..., Un, where X and Y are integers, where Z1 is an integer procedure (corresponding to the prime type $\sigma(T)$), where W1 is an integer procedure (corresponding to the prime type $T' \equiv \sigma(((T, \mathbb{Z}) \to T)) = ((T, \mathbb{Z}, T1, \ldots, Tn) \to \mathbb{Z})$), and U1, ..., Un are integer procedures (corresponding to the prime types T1, ..., Tn respectively). The algol procedure IND is related to the term t of Σ_0 as follows. Let a and b be integers, let f be a functional of type T, let g be a functional of type $((T, \mathbb{Z}) \to T)$, and let h1, ..., hn be functionals of types T1, ..., Tn respectively. Let p be the functional of type $\sigma(T)$ corresponding to the functional $(t; (a, b, f, g)) = ind(a, b, f, g)$ of type T. Let f1 be the functional of type $\sigma(T)$ corresponding to f, and g1 the functional of type T′ corresponding to g. Then

$$p(h1, \ldots, hn) = IND(a, b, f1, g1, h1, \ldots, hn).$$

Here is the algol 60 procedure IND.

```
INTEGER PROCEDURE IND (X, Y, Z1, W1, U1, ... , Un);
INTEGER X, Y; VALUE X, Y;
INTEGER PROCEDURE  Z1, W1, U1, ... , Un;
IF  X ≥ Y  THEN  IND := Z1(U1, ... , Un)  ELSE
                  BEGIN
                  INTEGER PROCEDURE  D(V1, ... , Vn);
                  INTEGER PROCEDURE  V1, ... , Vn;
                  D := IND(X, Y-1, Z1, W1, V1, ... , Vn);
                  IND := W1(D, Y-1, U1, ... , Un);
                  END;
```

The compilation of Σ into Σ_0 was originally suggested by the work of Gödel [2], but it is simpler and more natural to develop the compilation directly, as has been done here, without recourse to the "Gödel interpretation" of Σ. The missing ingredient in Gödel's paper, from our point of view, was the proof that the compilation algorithm does what we want it to do. According to Troelstra, this can be shown using a technique due to Kleene, which would involve an excursion into the Gödel interpretation. The method used here seems both direct and natural.

The language Σ_0 is concerned with just one aspect of formal mathematics-the communication of algorithms. The language Σ is concerned with a second aspect as well-the meanings of the algorithms being communicated and proofs that they work. The language algol 68, in addition to being concerned with the communication of algorithms, is concerned with yet a third aspect-the management of storage. One envisions some language, containing features of both Σ and algol 68, that concerns itself with all three aspects of formal mathematics. Of course, Σ is just a bare skeleton, and algol 68 may be too fully fleshed. It is interesting that the types of Σ are so similar to the types (called modes) of algol 68, but it is not clear that such a type structure will be adequate to the communication of general algorithms, such as arise in axiomatic theories (metric spaces, differentiable manifolds, measure spaces, etc.). It may be that something like a constructive version of Zermelo-Frankel axiomatic set theory will be more appropriate.

In addition to being a very general programming language, with sophisticated facilities for machine communication, our envisioned general algorithmic language (GAL) will hopefully be useful for human communication, i.e., for writing papers and textbooks. One can even envision the possibility of using a textbook,

say in conformal mapping, as a library of computer algorithms. Whether this possibility is realistic remains to be seen. The only attempt I know of to develop a completely formal language which is useful for human communication is due to A. P. Morse [4]. Of course, Morse's language is not constructive, so there is no question of using it as a programming language. It is hoped that the many features of the proposed language GAL would complement and enhance, rather than obstruct, one another.

REFERENCES

[1] E. Bishop, Foundations of Constructive Analysis, McGraw-Hill, New York (1967).

[2] K. Gödel, Über eine bisher nochnicht benützte Erweiterung des finiten Standpunktes, Dialectica 12 (1958) 280-287.

[3] S. C. Kleene, Introduction to Metamathematics, Van Nostrand, Princeton (1952).

[4] A. P. Morse, A Theory of Sets, Academic Press, New York (1965).

[5] S. G. van der Meulen and C. H. Lindsey, Informal Introduction to Algol 68, Mathematisch Centrum, Amsterdam (1969).