Cahiers du Centre de Logique
Volume 8

On the roles of types in mathematics

by

N.G. de Bruijn
Technological University Eindhoven

1 INTRODUCTION

When starting the AUTOMATH mathematics checking project in 1967, I almost immediately gave a very central position to the idea that is now often called 'propositions as types'. It came down to treating *proofs* the same way as *objects.*

Since logicians may wonder how such an idea dawned upon a mathematical mind without any training in logic, I shall try to explain here how natural that idea seemed to me. It will be a very personal account that will touch several aspects of the matter.

Very central in this is the PAL system for writing at least a limited form of mathematics. Having no lambda calculus facilities it will not be able to represent the bulk of modern mathematics, but yet this PAL was for me the core of all ideas about the various possibilities of the usage of typing. For more detailed information about PAL reference can be given to [2], [3], [5], [10], and for the AUTOMATH project in general to [5], [8].

Essential for starting a thing like PAL was the basic feeling that mathematical objects are typed, and that their typing should be an essential part of the mathematical language, since machines that check the mathematical texts should certainly know about the types.

Therefore this paper will start (sections 2 and 3) with some opinions, partially very personal, about the role of types in mathematics. Most of this was already given in [4].

The paper will also expose (in sections 15 and 16) some ideas about the use of typing that were not involved in the early AUTOMATH concept.

2 UNTYPED SETS

The very natural idea to attach a type to a thing seems to have hardly penetrated into today's ideas of the formal presentation of mathematics. Since about 1940 most mathematicians seem to claim that their work is based on Cantor's and Zermelo-Fraenkel's theory of untyped sets, combined with things like Boolean logic and Frege's predicate calculus. Whether this untyped set-theoretical basis is relevant for the style of their work in mathematics, is open to some doubt, however.

Working mathematicians do not care much for their foundation. This is not what non-mathematicians usually think: they expect that mathematicians are fully aware of the total structure of the building they are working at. Most mathematicians will say that their time is short, and that they have more important things to do than digging into foundations, but I think that this lack of interest is a sign that one is on the wrong track with the presentation of the basis of mathematics. And I think that an aspect that needs revision is the weird idea that *everything is a set.*

In the doctrine *everything is a set* the word *set* is clear. It is taken as *point in the Zermelo-Fraenkel universe* (to be abbreviated as ZF). But the words *everything* and *is* are harder. The word *is* does not seem to say that things *are* sets but that they can be *coded* as sets. It entirely depends on the coding system what particular sets are to be taken, and different people may have different codings. Theoretically, it seems to be perfectly legitimate to ask whether the union of the cosine function and the number e (the basis of natural logarithms) contains a finite geometry, but the question has no meaning at all as long as the codings have not been explicitly stated.

But how far should one go with *everything*? Is a theorem a set? Is a proof a set? And a proposition, an assumption, an axiom, a variable, a formula, a contradiction, a predicate, a geometrical construction? All these things can easily be coded in ZF, but most mathematicians will object to calling them sets. They would even object to talking about a set of theorems as a set!

Among mathematicians there seems to be a feeling that there is a world of mathematical objects with some Platonistic existence, and that these things are sets. And words like theorem, proof, variable, are considered to have a meaning in their discussion *about* those objects. They are not objects themselves. So not everything that can be coded in ZF is to be considered as an object. And what is worse, ZF itself is not an object.

The acceptance of ZF as a basis of mathematics is a social thing, and not more than that. The idea that ZF is interesting and important, is a self-fulfilling prophecy. Its more esoteric parts create many hard problems which challenge many clever mathematicians and logicians. But what is the use of it for the world outside? ZF may be an early example of a *black hole*, which is a cluster of matter in which mutual attraction is so immense that not even light is able to get out.

My personal opinion of ZF is a rather negative one. Cantor built a paradise by means of language that he did not properly define. Whatever he did was mixing language and metalanguage. This procedure was known to create several paradoxes, but as long as one did not come too close to the paradoxes, life seemed to be safe. The system was codified later by a set of restricting axioms which had as their main purpose to turn Cantor's shaky edifice into a solid sky scraper. One of the things that attracted people to ZF was the undeniable fact that well-ordering of sets like the continuum provided fast and elegant proofs for important theorems in ordinary analysis. And as usual in mathematics, elegance generates addiction: things that can be treated with the elegant method are called elegant and important, so the method becomes more and more indispensable.

On the other hand, if I have to be honest I have to admit that some of my feelings against ZF already hold for much simpler situations involving infinity. I find it hard to live in peace with the fact that it is undecidable whether there is a set of sets of natural numbers whose cardinality lies between the one of the set of all natural numbers and the set of all sets of natural numbers.

Why did the mathematicians so eagerly embrace untyped set theory around the middle of the twentieth century?

There were different kinds of people involved in this, of course. First there were the serious foundationalists, forming a kind of subculture. Once a subject is accepted in such a subculture and once it generates a considerable amount of serious work, it gets automatically important, and no further motivation is required.

Secondly, there were those involved in teaching mathematics on various levels. Of course, these people usually act on the signals of fashion too, but there is more to it. The general trend in mathematics was one of increasing rigour. But how can one give a rigorous treatment of the basis of mathematics without having to treat logic? The language of mathematics uses propositions and predicates, but mathematicians had no training in treating them seriously. One felt on much safer grounds when talking about mathematical *objects*. From a *predicate* one immediately passed to the *set* of objects satisfying that predicate. The predicates were treated quite informally, but once the transition to the sets was achieved, rigor could start.

Almost always the teacher as well as the students were somehow thinking in terms of typed sets, but no reference to types was ever made. I have never been able to understand what goes on in the minds of mathematical novices when confronted with untyped sets. I fear that in today's foundation-free mathematical education students just accept everything that is offered, rapidly training themselves in showing the behavior that is expected from them.

3 TYPED SETS

I believe that thinking in terms of *types* and *typed sets* is much more natural than appealing to untyped set theory. Here the word *natural* of course refers to our mathematical culture and not to the nature of things or to the nature of mathematics. In our mathematical culture we have learned to keep things apart. If we have a rational number and a set of points in the Euclidean plane, we cannot even imagine what it means to form the intersection. The idea that both might have been coded in ZF with a coding so crazy that the intersection is *not empty* seems to be ridiculous. If we think of a set of objects, we usually think of collecting things of a certain type, and set-theoretical operations are to be carried out inside that type. Some types might be considered as subtypes of some other types, but in other cases two different types have nothing to do with each other. That does not mean that their intersection is empty, but that it would be insane to even *talk* about the intersection.

A very clear case of thinking in terms of types can be found in Hilbert's axiomatization of geometry. He started by saying that he assumes there are certain things which will be called *points* and certain things to be called *lines*. Nothing is said about the nature of these things. In particular, no statement is made about lines being sets of points, and nothing expresses that these points and lines somehow live in a common 'plane'. In type-theoretical terminology one might formulate Hilbert's take-off as the introduction of two types, *point* and *line*, taken as primitives at the start of the theory.

Once the notion of a type is accepted, one can introduce typed sets by means of predicates running over such types.

Is there the drawback that working with typed sets is much less economic then with untyped ones ? If things have been said for sets of apples, and if these same things hold, *mutatis mutandis*, for sets of pears, does one have to repeat all what had been said before? No. One just takes a type variable, ξ say, and expresses all those generalities for sets of things of type ξ. Later one can apply all this by means of a single instantiation, replacing ξ either by *apple* or by *pear*.

I believe that the ordinary working mathematician pays lip-service to ZF, but still thinks in terms of typed sets. A superficial layer of formal training in ZF has not changed it. And I believe that much of the "New Math" rage of the middle of this century is caused by the lack of a generally accepted formal way to deal with predicates and types. The idea was that "sets" are no-nonsense objects with a kind of Platonistic existence.

Quite often there is no reason at all to use set terminology. And indeed, even quite rigid textbooks on analysis written in the first half of this century used the word set only when strictly necessary.

4 TYPES IN NATURAL LANGUAGE

My own feeling is that types are strongly related to the structure of some of the sentences in natural languages. Many English sentences contain the word *is* followed by an indefinite article (*a* or *an*), followed by a substantive or by a group of words playing the same role as a single substantive. Examples:

> n is an integer,
> n is a positive integer,
> the intersection of V and W is an n-dimensional space.

Let me call such sentences *typing sentences.* Note that instead of *is* one can have variations which play a similar role, like in *let x be a real number*, *P and Q are regular polyhedra.* Sentences with *are* can be ignored, since they can be considered as a contraction of *two* sentences, like *P is a regular polyhedron* and *Q is a regular polyhedron.*

Let me use the word *substantive* both for single substantives and for substantive groups. So in these examples *integer*, *positive integer*, *n-dimensional space*, *real number*, *regular polyhedron* are to be called substantives.

The group of words preceding the *is a* or *is an* will be called a *name.* So *the intersection of V and W* is a name. The name part of a typing sentence is intended to give a complete identification of a particular thing, and the substantive part of the typing sentence expresses what kind of a thing it is.

It should be noted that not all sentences containing the word *is* are typing sentences. In a sentence like *5 is the sum of 2 and 3* the word *is* indicates equality, and both *5* and *the sum of 2 and 3* have the form of a name. A more delicate case is *5 is the sum of two squares.* It is a contracted sentense saying that there exist two squares of which the sum is equal to 5. So *the sum of two squares* represents neither a substantive nor a name. On the other hand, replacing *the* by *a* one can claim that the sentence *5 is a sum of two squares* is a typing sentence, where *sum of two squares* is a substantive. The sentence certainly does not mean that there are squares a^2 and b^2 such that 5 is a sum of them.

For a detailed study of the grammar of the mixture of natural language and mathematical formulas I refer to [1].

5 NOTATION FOR TYPING SENTENCES

Let me use the colon for the *is a* (or *is an*), and write $A : B$ for the typing sentence *A is a B*. The colon notation will also be used for the declaration of a variable of a given type: the sentence *let x be a B* will be written as $x : B$. The distinction between *x is a B* and *let x be a B* will have to be expressed by other means, like the kind of place that the formula is given in the text.

6 EXPLAINING MATHEMATICS TO A MACHINE

In the next sections I shall indicate how the idea of treating proofs as objects is almost forced upon anyone who tries to explain mathematics to a machine.

All the mathematics presented to the machine is supposed to be written in a sequence of *lines*, which together form a *book*. The machine has to check the book line after line.

As a general principle I take it that the checking machine is never expected to make an undirected search through previous parts of the book for material it needs. It is the task of the human writer to provide the references, and these references are to be considered as an essential part of the text.

7 EXPLICIT DEFINITION OF A FUNCTION, AND APPLICATION OF SUCH A DEFINITION

In modern analysis many functions are defined by means of operators acting on known functions, but in more old-fashioned mathematics functions were introduced only by explicit description of the function value for 'arbitrary' values of the argument or arguments. The following example describes the definition of a real-valued function with values $f(x,w)$, where x is a real variable and w a complex one. One begins by saying *let x be a real number*. This means that one has opened a *context* in which the letter x is treated as if it were a very particular real number.

A number of sentences to follow will be said in this context. In all those sentences the x may occur. Whatever is said in this context is completely independent of what particular number is represented by x. Therefore, everything that is said may immediately be repeated later, with the x replaced by some particular real number.

Inside the context of x one may open a new context by means of *let w be a complex number*. In that context of x and w one may build an expression E containing both x and w, and then it is possible to define a function by saying that its value $f(x,w)$ is given by E. This has the effect of defining the function for all values of the variables, since it is allowed to replace the x and w later by expressions A and B representing a real number and a complex number, respectively, where it is agreed that $f(A,B)$ is equal to what the expression E becomes if x is replaced by A and w by B.

In this setting it would be confusing to say that f is the function. I rather call it the *function identifier*.

The sentences *let x be a real number* and *let w be a complex number* will be called *declarations of a (typed) variable*.

It should be remarked that the expressions A and B need not represent completely defined numbers: the function call $f(A,B)$ may take place inside some other context containing a number of variables, and the A and B may be expressions containing those variables. The function call might even take place in the context of x and w itself, with A and B being expressions containing x and w.

In the context of the first variable x, one can also consider expressions containing x and representing *types*. Having some type $T(x)$ one can start a new context (inside the one of x) by saying *let w be something of type $T(x)$*. Example: *let n be a natural number* and *let s be a set of n real numbers*. Again it is possible to define a function value, $g(n, s)$ say, inside this context. The matter of a later function call $g(A, B)$ is now slightly more complicated. In order to legitimate this call it has to be verified that A is a natural number and that B is a set of A real numbers. So all the time the machine has to check typings.

8 TELESCOPE TERMINOLOGY

The following notation can be used for handling the composite contexts discussed in the previous section. The context *let x be a real number and let w be a complex number* can be abbreviated as $[x : R][w : C]$ (where R and C represent the real and the complex type). In the case where the type of the second variable contains the first variable one has something like $[x : P][w : Q(x)]$. Such a string is called a *telescope* (irrespective of whether it is taken to represent a context or not). The word was inspired, of course, by the old-fashioned optical instrument consisting of segments that slide one into another.

If in a context $[x : P][w : Q(x)]$ a function value $f(x, w)$ is defined, a later call $f(A, B)$ can be made if $A : P$, $B : Q(A)$. This pair of typings will be expressed by saying that the vector (A, B) *fits into* the telescope $[x : P][w : Q(x)]$.

This terminology will be used likewise for longer strings. If in the context

$$[x_1 : P_1][x_2 : P_2(x_1)][x_3 : P_3(x_1, x_2)] \cdots [x_k : P_k(x_1, \ldots, x_{k-1})] \qquad (1)$$

a function value $f(x_1, \ldots, x_k)$ is defined, then a function call $f(A_1, \ldots, A_k)$ can occur later, provided that the vector $(A_1, \ldots, A_k)$ fits into the telescope, which means that

$$A_1 : P_1,\ A_2 : P_2(A_1),\ A_3 : P_3(A_1, A_2)\ , \ldots, A_k : P_k(A_1, \ldots, A_{k-1}).$$

Forming the function call $f(A_1, \ldots, A_k)$ is called *instantiation*.

9 BOOKS IN PAL

PAL is a primitive form of AUTOMATH. Actually it is that part of AUTOMATH that does not use the lambda calculus. In spite of its primitivity, it is able to represent the structure of mathematics with objects, types, contexts, definitions, lemmas, theorems and axioms. Adding lambda calculus to PAL is a step that requires quite some technicalities, but it need not be discussed here, since the conceptual aspects of what mathematics is, and what a machine can check, are largely covered by PAL.

Books in PAL are sequences of lines. Every line is written in a *context*, which has the form of a telescope. Usually one thinks of the context being constant over a number of consecutive lines, and one usually thinks of only small changes in context, where passing from a context of a line to the context of the next line means either adding one item $[\cdots : \cdots]$ at the far right end of the telescope or just deleting the last item of the telescope.

The lines of the book all have more or less the same form. The standard case is this one:

$$f \ := \ A \ : \ B.$$

The interpretation is that the A is a (usually composite) description of an object, B is its type, and the (totally fresh) identifier f is a new name for A. One can also say that f is an *abbreviation* for A, or that *f is defined* by A. The line as a whole can be called a *definition*, or a *definitional line.*

The cases of one-item extensions of the context are interpreted as *declarations.* They give an obligation: if a context (1) has to be extended by $[x_{k+1} : P_{k+1}(x_1, \ldots, x_k)]$ then it has to be required that in the context (1) itself the expression $P_{k+1}(x_1, \ldots, x_k)$ is a well-formed type.

It is easy to see how a PAL book can be continued once it is under way. But in order to get it started, the following features have to be added: primitive objects, type variables, type definitions and primitive types.

An example of a *primitive object* is the introduction of the number 1 in the Peano axioms. Taking it for granted that there is a type called *natural number* already, one proceeds by saying that there is a particular natural number that will be called "1", which will be treated as a known object. It is in no way stated what it is: it is not given by an expression in terms of older known objects. So in a way this 1 looks like a variable, but it is not. The 1 is not obtained by a context extension, and it is never allowed to replace 1 by any other natural number, like it would have been if it were a variable.

In PAL the introduction of a *primitive object* of type T is written as a book line

$$p := \text{PN} : T,$$

written in the empty context. It is called a *primitive line.* The symbol PN is not to be manipulated like other expressions: it just has to stand there in order to let the line look like an ordinary definitional line. In a way the primitive line is treated as definitional line: the identifier p introduced here will be a legitimate symbol of type T from now on. A definitional line like $q := S : T$ is different from a primitive line in the sense that in later lines one has the option to replace q by its definition S. But in case of a line $p :=$ PN : T the PN gives no answer to the question what p is.

It is also possible to write a primitive line in a non-empty context. The line $f :=$ PN : T in the context $[x : A][y : B(x)]$ can be interpreted as the introduction of a primitive function of the variables x and y. In later use the f will be available for instantiation with a vector (K, L) fitting into the context, just as if the line had been a function definition. As an example one can take the introduction of the successor of a natural number in Peano's axioms. In the context *let x be a natural number* the successor of x is taken as a primitive, and the primitive line expresses that this successor is again a natural number. The instantiation mechanism sees to it that the successor is treated as a function. And again, the PN gives no answer if one wants to know what the value of the function is exactly.

The issues *type variables*, *primitive types* and *type definitions* have in common that they handle types as if they were objects.

With the declaration of a type variable the context is extended. In natural language one would say *let ξ be a type.* If ξ is a type then it can be used as a type for ordinary objects: one might say *let ξ be a type and let x be a ξ.*

In the declarational sentence *let ξ be a type* the word *type* stands at the place of the substantive, but one has to be careful: this *type* is not a type. In order to try to avoid confusion the word *type* will be printed bold face when written in the book. So a context may be extended by $[\xi : \textbf{type}]$.

The matter of primitive types is now clear. One just writes a line like $\eta :=$ PN : **type**. This legimates the further use of η as a type. If the line is written in a non-empty context, it introduces a primitive type-valued function of a number of variables (which may now be either object variables or type variables).

Finally it can be said what type definitions are. They have the form $\zeta := \Theta : \textbf{type}$, where the typing $\Theta : \textbf{type}$ has already been established. It is nothing but an abbreviation of Θ to a new symbol ζ. If the abbreviation is given in a non-empty context, then it describes ζ as a function of a number of variables.

The standard way to get such expressions Θ is by instantiation of functions that were either defined or proclaimed as primitive before.

With all these provisions, one can start off writing books in PAL.

Let me explain the notion of *degree* here. The term **type** is said to have degree 1. If ξ is a type then ξ is said to have degree 2. If moreover $x : \xi$ then x is said to have degree 3.

Lines in a PAL book are either "$f := S : T$" or "$f :=$ PN : T". Such lines are called *lines of degree 2* if T is **type** (and in that case f and S have degree 2), and *lines of degree 3* if T has degree 2 (and in that case f and S have degree 3). So the degree of a line is the degree of the identifier in front of the := sign.

Accordingly, a context item $[x : T]$ is said to have degree 2 if T is **type**, and to have degree 3 if T has degree 2. So the degree of a context item is the degree of its variable.

Summarizing, in PAL the lines are either of degree 2 or 3, and they are either definitional or primitive (PN-lines). Contexts can be mixed strings of items of degrees 2 and 3.

The rule for the fitting of a vector into a telescope can be formulated as before, without even mentioning the degrees.

Here is an example of a short PAL book. It represents the beginning of the Peano axiom system, and the definition of the numbers 2 and 3. And, just for fun, a double successor is defined, and the number 4 is introduced by means of it. In this layout there is an asterisk in front each line, and the context telescope is printed to the left of it.

	⋆	nat	:=	PN	:	**type**
	⋆	1	:=	PN	:	nat
[x:nat]	⋆	succ	:=	PN	:	nat
	⋆	2	:=	succ(1)	:	nat
	⋆	3	:=	succ(2)	:	nat
[x:nat]	⋆	doubsucc	:=	succ(succ(x))	:	nat
	⋆	4	:=	doubsucc(2)	:	nat

I will not try to describe all the rules of the game here. Once the *interpretation* is understood, it will be pretty obvious what the rules have to be. I only want to say here that the notion of *definitional equality* is vital. Two expressions are called definitionally equal if they can be reduced to a common one, where reduction is taken in the sense of repeated application of book definitions, and here *application of* $f := A : B$ means, roughly speaking, replacing all f's by A. An example in the text above is that 3 is definitionally equal to doubsucc(1).

Definitional equality plays a role when a typing like $P : Q$ has to be checked. The type of P can be evaluated algorithmically, and then the question remains whether this evaluated type is definitionally equal to Q.

10 THEOREMS COMPARED TO DEFINITIONS OF FUNCTIONS

Section 7 discussed the definition of a function of several variables by means of the description of its value inside a (telescopic) context. Such a piece of text might be called a *blueprint:* any application of that definition can be seen as copying the text of the definition, replacing the variables by specific values.

A *theorem* can be considered as a kind of blueprint too. Quite often a theorem emerges from the fact that a piece of mathematics shows some repetitions. There can be a number of fragments which are very much alike, and in such cases mathematicians rapidly discover that they can write a single blueprint of which all those fragments are copies, merely adapted to the local situations. This is parallel to what is done with functions. Instead of defining the logarithms of 3, 4 and 5 separately, the logarithm of the real variable x is defined, and the logarithms of 3, 4 and 5 can be found by taking that definition and replacing all x's first by 3, then by 4, and then by 5.

The description of a *theorem* is similar to the one of a function. In general, a *mixed* context has te be taken. Not all segments that build up the telescopic context are declarations of variables. Some will be *assumptions*. These two different kinds of segments will be called *declarational* and *assumptional* context extensions, respectively.

Here is an example of a theorem:

> *Let n be a natural number, let w be a complex number and ε a positive real number. Assume that $|w^n| < 1$. Then there exists a positive integer k such that $|w^k| < \varepsilon$.*

A proof of this theorem will have the following form. Inside the context formed by the declarations of the variables n, w and ε a new context is opened by the assumption $|w^n| < 1$. Throughout that context the letters n, w and ε are considered as natural, complex, and positive real numbers, respectively, and the the assumption $|w^n| < 1$ is considered as a true statement from which further truths can be derived. And in that context it is shown that there exists a positive integer k such that $|w^k| < \varepsilon$.

What has to be done in order to *apply* that theorem later? Then there are, in some context or other (let it be called the *new* context), expressions A, B, C, representing a natural number, a complex number and a positive real number, respectively. And in that context it is known that $|B^A| < 1$. By application of the old theorem one concludes that there exists a positive integer k such that $|B^k| < C$.

How does one explain to a machine that this application of the theorem is legitimate? It is easy to let the machine check whether the substitutions satisfy the rules. And, though less easy, the very essential matter of type-checking (here for the types of A, B, C) can be left to a machine too. But how should the machine be convinced that $|B^A| < 1$ is satisfied?

One has to require that $|B^A| < 1$ is not only *true* in the new context, but that it has already been stated and proved earlier in the book. If in a mathematical argument one wants to use some particular proposition, a mere appeal to the fact that the proposition is true will not do. It should have been proved somewhere in the past, or else it should have been introduced in an assumption of the context.

According to the general principle explained in section 6, the application of our theorem should somehow mention a reference to the place where this $|B^A| < 1$ was proved or assumed. In what form should such a reference be given?

The statement of the theorem was given in a mixed context of declarations (of typed variables) and assumptions. If it is only a matter of declarations, the full context of the theorem can be written as a telescope, and the theorem is applied by instantiation, i.e., by giving a vector that fits into that telescope. Can the references to proofs that *assumptions* are valid in the particular application, be expressed by means of that fitting mechanism as well? The matter of assumptions and proofs of their validity has some of the characteristics of the instantiation already. Can it be brought in exactly the same form?

It may help to visualize the theorem application as a kind of dialogue between mathematician and machine. The mathematician claims a certain application, and the machine asks questions about if. Let the theorem be formulated as a statement $T(x, y, z)$ made in the context of a variable x of type P_1, an assumption $A_1(x)$, a variable y of type $P_2(x)$, an assumption $A_2(x, y)$, a variable z of type $P_3(x, y)$ and an assumption $A_3(x, y, z)$. What questions will the machine ask if the mathematician claims an application of the theorem? The following dialogue (in which X, Y and Z are expressions, not necessarily variables) might take place.

MACHINE. — Give me something of type P_1 that can play the role of x.

MATHEMATICIAN. — X.

MACHINE. — OK, I see that $X : P_1$. Next I have to check the validity of the assumption $A_1(x)$, but it has to be updated. So convince me of the validity of $A_1(X)$.

MATHEMATICIAN. — Formula (27).

MACHINE. — OK. Now give me something of type $P_2(X)$ that can play the role of y.

MATHEMATICIAN. — Y.

MACHINE. — OK. I see that $Y : P_2(X)$. Updating $A_2(x, y)$, I want to be convinced of the validity of $A_2(X, Y)$.

MATHEMATICIAN. — Formula (21).

MACHINE. — OK. Now give me something of type $P_3(X, Y)$ that can play the role of z.

MATHEMATICIAN. — Z.

MACHINE. — OK, I see that $Z : P_3(X, Y)$. Now convince me of the validity of $A_3(X, Y, Z)$.

MATHEMATICIAN. — Formula (24).

MACHINE. — OK, I now accept the theorem application $T(X, Y, Z)$ as having been proved.

Instead of the answer *Formula (24)* the situation is often slightly more complicated. Instead of this reference, it might be a *composite reference*, like a reference to a formula (18) that indicated a statement $Z(u, v)$ given in a declarational context *let* $u : U,\ v : V$. And the mathematician has two expressions K and L such that the vector (K, L) fits into that context $[u : U][v : V]$. Instead of the reference *Formula (24)* he might have given *Formula (18), with u and v replaced by K and L*. It looks like instantiation of the vector (K, L) into a function, where the reference *Formula (24)* plays the role of the function identifier. The machine has to check that (K, L) fits into the telescope $[u : U][v : V]$, and that $Z(K, L)$ is the same thing as $A_3(X, Y, Z)$.

This matter of composite references is comparable to ordinary function calls in cases where the instantiations are made by means of expressions which are instantiations themselves, like in $f\big(g(x, y), y, h(x)\big)$.

It is clear that a reference like *Formula (21)* is not a reference to a proposition but to a place where that proposition is proved. One might say that this proof reference is a *name* for a proof. It is of the same kind as a name for a mathematical object and is manipulated similarly.

In the dialogue the machine required the mathematical expressions to have the right type. Similarly it requires something from a proof reference like

Formula (27): this has to indicate a proof of the proposition $A_1(X)$. And the machine wants to check that the place referred to indeed proves exactly that proposition. This will be just another case of type checking.

11 THE VARIOUS ASPECTS OF THE PARALLEL

To start with, it helps to express the situation in natural language. Sometimes the machine has to to check that a certain thing *is* a rational number, sometimes it has to verify that a certain piece of text *is* a proof of a certain statement. The machine wants to convince itself that formula (27) is a proof of $A_1(X)$. So *Formula (27)* is grammatically a *name*, and *proof of* $A_1(X)$ is a *substantive.*

The parallel now becomes clear. Proofs correspond to objects, and have to get names that are manipulated in the same way as the (often composite) names of objects.

Assumptions correspond to declarations. Let me compare *assume* P (where P is some proposition) to the declaration *let* x *be an* A. The declaration contains the type information A as well as an identifier x that will be treated as if it were an object, always the same object throughout the context opened by the declaration. In order to treat the assumptional contexts similarly, one has to introduce an identifier like u, and to interpret *assume* P as *let* u *be a proof of* P. This u can be used throughout the context as if it *were* a proof of P. Just like the x above, the u is available all through the context, and is not to be used outside.

Having a proof of the proposition P is to be taken as having something of type *proof of* P. In particular, if in some context γ there is a line

$$\gamma \star h := \Theta : \text{proof of } P$$

then that means that P is proved in that context. The context may be a sequence of declarations and assumptions, and in applications of the theorem a fitting vector should be produced.

All this reveals the full parallel between function definitions (as in section 7) and theorems.

And what is an axiom? An axiom is like a theorem, but it just *pretends* that there is a proof instead of giving a proof. It is completely parallel to the introduction of a primitive object (see section 9). If one wants to proclaim the proposition Q as an axiom in some context γ, one writes a line

$$\gamma \star \alpha := \text{PN} : \text{proof of } Q.$$

The text presented to the machine does not mention *names* of theorems like in ordinary mathematical writings. Instead, it handles names of *proofs*, which is rather unconventional. And in the case of an axiom it handles the name of a pretended proof. In all respects the name of the proof plays the same rôle as the name of an object.

When exploiting the object-proof parallel the way described here, one can say that mathematics is nothing but handling abbreviations. Abbreviations for composite names of objects are piled up, and so are composite names of proofs.

It is also due to these abbreviations that the proof of an ordinary mathematical theorem can be written in a single short line. All the previous material of the proof has meanwhile been abbreviated, and can be referred to by means of short expressions in the final proof line.

But how do get things like *proof of Q* into our PAL book in the first place? One way to do it is by means of the introduction of a function *proof* acting on propositions. There is a primitive type called *propositon*, and saying that *P: proposition* of course means that P is a proposition. It was the AUTOMATH tradition to write *bool* (for *boolean*) instead of *proposition*, and with that notation the text goes like this:

$$\begin{array}{rclcll} & \star & \text{bool} & := & \text{PN} & : & \textbf{type} \\ [b{:}\text{bool}] & \star & \text{proof} & := & \text{PN} & : & \textbf{type} \end{array}$$

So from now on the substantive *proof of P* gets into the book as the type *proof(P)*. If u is a proof of P, one writes *u:proof(P)*. So *proof(P)* has degree 2, and u has degree 3.

Two different kinds of lines will appear in the book: object lines and proof lines. The machine does not have to bother of what kind the lines are, since it treats both kinds the same way. If the middle part of the line is not PN, the line is called a definition (or abbreviation) in the object case, and a theorem (or lemma) in the proof case. If the middle part is PN, the line is the introduction of a primitive object (mathematical tradition does not seem to have coined a word for such lines) in the object case, and an axiom in the proof case.

Similarly, there are two kinds of context items, the declarational and the assumptional ones.

12 HEYTING'S INTERPRETATION OF IMPLICATIONS

It was only after having formed all these ideas about the use of PAL in 1967, that I recalled discussions I had with A. Heyting in the fifties (I was a close colleague of Heyting at the University of Amsterdam from 1952 to 1960). He considered a proof of an implication $P_1 \to P_2$ as a kind of procedure that, when given a proof of P_1, was able to get a proof of P_2. In other words, it is a kind of function that maps proofs of P_1 into proofs of P_2. In the latter version, one might say that a proof can be a function argument as well as a function value, and that suggests that proofs are considered as objects. But I do not think that Heyting would have formulated it that way.

Whether this idea played a role on the back of my mind when developing PAL, I really do not know.

In PAL such a proof of an implication follows this scheme:

$$\begin{array}{lllll} & \star\ P_1 & := & \cdots : & \text{bool} \\ & \star\ P_2 & := & \cdots : & \text{bool} \\ [u : \text{proof}(P_1)] & \star\ v & := & \cdots : & \text{proof}(P_2) \end{array}$$

The notion of *implication*, as a function of two boolean variables, can be introduced as a primitive in PAL:

$$\begin{array}{lllll} [b_1 : \text{bool}][b_2 : \text{bool}] & \star\ \text{impl} & := & \text{PN}: & \text{bool} \end{array}$$

But if one wants to express in PAL, albeit by some kind of axiom, that the above Heyting-type implication proof v leads to some line

$$\star \quad w \quad := \quad \cdots : \quad \text{proof}(\text{impl}(P_1, P_2))$$

then this requires lambda-abstraction, a thing that PAL does not provide. The situation is the same as with functions. If in PAL one gives the explicit definition of some function that maps objects of type A to objects of type B, then the fact that it is a function from A to B is *metalanguage* of PAL. There is no way to derive in PAL that this definition leads to an object of which the type is *mapping(A, B)*.

There is the similar situation with universal quantification. In PAL one can write that for every x of type A there is a proof of the proposition $P(x)$, just by a line $[x : A] \star u := \cdots : \text{proof}(P(x))$, but it requires lambda abstraction to write (stepping out of the context $[x : A]$) that there now is something of a type like $\text{All}(A, P^\star)$ (where $P^\star$ is a predicate corresponding to the function identifier P).

The great similarity between functional abstraction, universal quantification and derivation of implication certainly played a role in inventing the proofs-as-objects philosophy.

13 WRITING DIRECTLY IN TERMS OF PROOF TYPES

As said before, in a PAL book there are two kinds of lines: object lines and proof lines. They are treated alike, but there is still a lack of symmetry. In the object world there were lines of degrees 2 and 3, but thus far the lines in the proof world have degree 3 only. The only exception was the line $[b : \text{bool}] \star \text{proof} := \text{PN} : \textbf{type}$ of section 11. And thus far the expressions of degree 2 in the proof world all had the form *proof (P)*. But if one starts from a type variable, there is no way to say that it has the form *proof(P)* where P is a proposition. Already in an early stage of the AUTOMATH project this was one of the reasons why a second term of degree 1 was adopted. It was called **prop**, but I now think that something like **prtype** might have been a better notation.

This **prop** plays the same role in the proof world as **type** plays in the object world.

With **prop** one can introduce proof types without saying to what propositions they belong. When writing mathematics one hardly ever has to mention propositions: everything can be done directly in terms of the proof types. Writing in this fashion is called **prop**-style (in contrast to taking the propositions themselves as basic, what is called bool-style).

For example, conjunction and axioms for conjunction can be introduced directly in terms of the proof types:

$[\xi:$**prop**$][\eta:$**prop**$]$	$\star$	CON	:=	PN	: **prop**
$[\xi:$**prop**$][\eta:$**prop**$][u:\xi][v:\eta]$	$\star$	AX1	:=	PN	: $\mathrm{CON}(\xi,\eta)$
$[\xi:$**prop**$][\eta:$**prop**$][w:\mathrm{CON}(\xi,\eta)]$	$\star$	AX2	:=	PN	: ξ
$[\xi:$**prop**$][\eta:$**prop**$][w:\mathrm{CON}(\xi,\eta)]$	$\star$	AX3	:=	PN	: η

The idea of propositions and proof types of propositions can be added nevertheless. It is the same thing as in section 11, with the only difference that in the second line there has to be **prop** instead of **type**:

	$\star$	bool	:=	PN	:	**type**
$[b$:bool$]$	$\star$	proof	:=	PN	:	**prop**

Discovering the possibility to write in **prop**-style is just another case of an idea suggested by analogies in formal notation.

14 OTHER USES FOR THE PAL SYSTEM

Thus far I considered books where two kinds of lines, object lines and proof lines, are interwoven, and where the context telescopes can be mixed as well, forming sequences of declarations and assumptions. It is natural to ask whether there are more than those two opportunities.

Indeed there are. Historically, the first thing that should be mentioned is the tradition of geometrical constructions with ruler and compass. Just like the situation with objects and proofs, constructions can be expressed as instantiations of simpler constructions. The book can describe such constructions, and can express that $\cdots$ *is a construction of* P, where P is some geometrical point. This phrase contains the typical *is a*, and therefore *construction of* P can be considered as a substantive. It is a parametrized substantive, of the same kind as the *proof of* P where P was a proposition.

Accordingly, one has the same options as with *bool* and **prop**. Either one starts with "construction(P):**type**", or one takes a new basic expression **constr** of degree 1, comparable with **type** and **prop**.

Primitive lines "c := PN : construction(P)" can be seen as the proclamation of the possibility of fundamental constructions (like taking the intersection of two lines that have been constructed already). And what would a context item *let* C *be a construction of* P mean? It means a pretended construction, that is, the point P is just falling out of the blue air. In ordinary words, it means *take an arbitrary point* P.

It is a very subtle interplay between geometrical objects, proofs and geometrical constructions. There are various ways to play the game in PAL. I refer to [6] for details.

Another thing that might be mentioned is to handle two or more different notions of proof in a mathematical text. One might take as basic expressions of degree 1 **prop** for classical proofs and **intprop** for intuitionistic proofs. A book might contain both kinds of proofs, and the two might support each other. It is a bit similar to having, in the book of geometrical constructions, descriptions of constructions with ruler alone along with the ordinary ones. Constructions with ruler and compass might make use of earlier constructions with ruler alone, and the other way around.

The art of geometrical constructions is about constructions with a pencil on paper with ruler and compass, and all these are idealized. Similarly, theoretical computer science is about programs executed by an idealized computer.

And indeed, the matter of syntax and semantics of computer programs shows much of the old spirit of the Greek geometry. It describes constructions (computer programs) and proves that they actually achieve what is promised in the program specifications. One can try to arrange that the mathematics, the logic, the program description and the program correctness proofs are all put into one and the same book, which can be checked in just a single run. I refer to [7] for this.

15 GENERALIZATIONS OF PAL

PAL can be generalized by adding new features, allowing what was not allowed before. The most important one among such features is of course the lambda calculus, but I already said I want to ignore that here since most ideas about the roles of typing can be explained in a lambda-free setting.

A quite obvious extension of PAL is allowing more degrees than the usual degrees 1,2,3.

In section 14 there appeared the beginning of an avalanche of expressions of degree 1: **type**, **prop**, **intprop**, and **constr**. One might dislike that such things have to be declared beforehand in the language definition. One might feel that the right to introduce such expressions should be delegated to the user, just like the user already had the right to introduce things of degree 2, like *nat* and *bool*.

This is easily achieved by allowing expressions of degee 0, and adopting a symbol **supertype** of degree 0. Consequently, the symbols **type**, **prop**, etc. are no longer needed in the language definition. The user can introduce them in book lines:

```
⋆  type    :=  PN  :  supertype
⋆  prop    :=  PN  :  supertype
⋆  constr  :=  PN  :  supertype
```

Such lines are not necessarily restricted to the beginning of the book. They can also be written later. In particular they may be given in a context, like this:

```
[n:nat] ⋆  typ   :=  PN  :  supertype
```

which gives the right to use infinitely many expressions of degree 1, obtained by instantiation:

$$\mathbf{typ}(1),\ \mathbf{typ}(2),\ \mathbf{typ}(3),\ \ldots$$

16 GIVING UP UNIQUENESS OF TYPES

As mentioned at the end of section 9, the PAL system maintains the principle of *uniqueness of types.* Roughly speaking, this has the effect that if somewhere in the book a typing $A : B$ is valid, and somewhere else $C : D$, where A and C are definitionally equal, then B and D are definitionally equal too.

The main reason for this uniqueness requirement was the general idea (cf. section 6) that the machine should be able to check the book without any further hints. But this is not a serious objection, since there might be good notational system for such hints, and as such they might become part of the language. After all, PAL has this nature itself already: a substantial part of a PAL book consists of things that would be considered as informal hints in standard mathematics.

When typing is seen as an *is a* relation between a name and a substantive (section 4), there is no need to require that relation to be a *function* from names to substantives, where the substantive is always the uniquely determined function value. The ordinary usage of *is a* does not require this. One can say that *3 is a positive rational* and *3 is a non-zero complex number*, without requiring that the substantives *positive rational* and *non-zero complex number* are definitionally equal.

In an example like this one can try to explain things in terms of *subtypes.* One can claim that there is a more universal type like *number* of which both *positive rational* and that *non-zero complex number* are subtypes. In [1] I called such a universal type an *archetype.* Subtypes can be described by means of predicates on the archetype. The archetype of an object is unique, and by means of a simple system of hints the checking machine is able to find that archetype. It only remains to check that the object satifies the predicate that belongs to the subtype in question.

So uniqueness is not really given up in this subtype system, it is just hidden. But one might think of a modification of PAL where things can have several different types which are *not* embedded in a common archetype. One can think of admitting book lines which express that some p is defined as E and typed both by A and B. And one might think of context items of the form *let x be of type C or D*. The substitution and instantiation mechanism of PAL would easily handle this kind of thing. The question is only whether it would ever be of any use for the mathematical sciences.

Non-uniqueness seems to be more essential if the notion of typing is generalized to the fitting of vectors in telescopes.

Mathematicians use *is a* for building sentences which can not be directly interpreted as typings in PAL. In the sentence *let G be a group* the word *group* cannot be related to just a type. And the G is more than a simple name. What G stands for is a vector, which may contain (depending on the taste of the user) a type, a predicate on that type (together determining a set), a ternary relation *product* on that set, a unit element, and finally a number of entries providing proofs for the group properties.

It is possible to extend PAL to a language in which names (like *group*) are given to telescopes. Such names can be introduced in a new kind of book lines on the left of the $:=$ symbol. A corresponding facility is the abbreviation of vectors of arbitrary length by means of a single symbol. Both kinds of abbreviations can be given inside a context, and then the instantiation mechanism can be applied to form new telescopes and new vectors. In such a language enriched with telescopes and vectors the fitting of a vector into a telescope can be seen as a generalization of typing (but in order to get things straight this requires that ordinary typing $p : T$ has to be rephrased as $p : [x : T]$) .

Around 1974 J. Zucker (cf. [11]) wrote a considerable fragment of mathematics in terms of AUT-Π with the use of abbreviations for vectors and telescopes, directly corresponding to the colloquial usage in standard mathematics. This AUT-Π is a lambda calculus extension of PAL, but the spirit of telescope handling is independent of lambda calculus.

In [9] I described how mappings of a telescope into another one (like a function that attaches a field to every abelian group) can be treated by a calculus that is completely analogous to the treatment of ordinary mappings from some type into some other type.

I quote this matter of telescopes here since it is quite workable in spite of the fact that the analog of uniqueness of typing fails. Zucker gave a simple example that kills it convincingly: if (v_1, v_2) fits into the telescope $[x : P][y : Q(x)]$, then it also fits into $[x : P][y : Q(v_1)]$, and these two telescopes can in no way be considered as definitionally equal.

17 EPILOGUE

From the way I exposed the idea of *proofs as objects* in the central sections of this paper the reader will understand that I find the term *propositions as types* somewhat misleading.

If p is a proposition, then the parallel is as follows. In the object world there are the typings of degree 3 and 2, respectively:

$$3 \ :\ \mathrm{nat}, \quad \mathrm{nat} \ :\ \mathbf{type}$$

In the proof world there are corresponding typings:

$$u \ :\ \mathrm{proof}(p), \quad \mathrm{proof}(p) \ :\ \mathbf{prop}$$

So it is not the proposition p itself that plays a role similar to the one of the type nat, but proof(p). When writing **prop**-style, this is not different. Then there are typings like

$$u \ :\ P, \quad P \ :\ \mathbf{prop}$$

and P is not a proposition but a particular kind of substantive for which I know no grammatical term.

In the **prop**-style writing tradition of the AUTOMATH project it was customary to use identifiers that were more or less copies of standard names for the corresponding propositions. This was also done in section 13 above, where the identifier "CON" was chosen to play the following role: if $u : \mathrm{CON}(\xi, \eta)$ then u is a proof of the conjunction of the two propositions of which ξ and η are proofs. So "CON" is related to the proposition that would normally be denoted by "con" but it is certainly not to be identified with it.

A related remark is that I would not like to identify a type with a class. In a natural language like English there is a clear distinction between a substantive (like *bottle*) and the corresponding class (*the class of all bottles*). The substantive is the generic name for the class. When abbreviating substantive and class by S and C, respectively, it is reasonable to identify C with *the class of all S's*, and S with the substantive *element of S*. But S and C should not be identified.

REFERENCES

[1] N.G. de Bruijn. The Mathematical Vernacular, a language for mathematics with typed sets. In *Proceedings of the Workshop on Programming Logic*, University of Göteborg and Chalmers University of Technology. Report 37, Programming Methodology Group, ISSN 0282-2083.

[2] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration (Versailles, December 1968). Lecture notes in Mathematics, Vol. 125*, pages 29–61. Springer Verlag, 1970.

[3] N.G. de Bruijn. AUTOMATH, *a language for mathematics.* Séminaire Math. Sup. 1971. Les Presses de l'Université de Montréal, 1973.

[4] N.G. de Bruijn. Set theory with type restrictions. In A. Hajnal, R. Rado, , and Vera T. Sos, editors, *Infinite and Finite Sets (volume I)*, pages 205–314. Coll. Math. Soc. J. Bolyai, vol. 10, 1975.

[5] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in combinatory logic, lambda calculus and formalism*, pages 579–606. Academic Press, 1980.

[6] N.G. de Bruijn. Formalization of constructivity in AUTOMATH. In P.J. de Doelder, J. de Graaf, and J.H. van Lint, editors, *Papers dedicated to J.J. Seidel*, pages 76–101, Eindhoven University of Technology, 1984. EUT-Report 84-WSK-03, ISSN 0167-9708, Department of Mathematics and Computing Science.

[7] N.G. de Bruijn. The use of justification systems for integrated semantics. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog-88, International Conference on Computer Logic Tallinn USSR, December 1988*, pages 9–24. Lecture Notes in Computer Science, vol. 417, Springer-Verlag, 1990.

[8] N.G. de Bruijn. Checking mathematics with computer assistance. *Notices American Mathematical Society*, 8(1):8–15, January 1991.

[9] N.G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91:189–204, 1991.

[10] D.T. van Daalen. *The language theory of* AUTOMATH. PhD thesis, Dept. of Mathematics, Eindhoven University of Technology, 1980.

[11] J. Zucker. Formalization of classical mathematics in AUTOMATH. In Guillaume, editor, *Colloque International de Logique, Clermont-Ferrand*, pages 135–145, Paris, 1975. Editions du Centre National de Recherche.

Technological University Eindhoven
Department of Mathematics and Computing Science
PO Box 513
5600 MB Eindhoven, The Netherlands