A Survey of the Project Automath*

N.G. de Bruijn

1. PURPOSE OF THIS SURVEY

Thus far, much about Automath has been written in separate reports. Most of this work has been made available upon request, but only a small part was published in journals, conference proceedings, etc. Unfortunately, a general survey in the form of a book is still lacking. A short survey was given in [de Bruijn 73c], but the present one will be much more extensive. Naturally, this survey will report about work that has been done, is going on, or is planned for the future. But it will also be used to explain how various parts of the project are related. Moreover we shall try to clarify a few points which many outsiders consider as uncommon or even weird. In particular we spend quite some attention to our concept of types and the matter of "propositions as types" (Section 14). Finally the survey will be used to ventilate opinions and views in mathematics which are not easily set down in more technical reports.

Some further material of a general nature can be found e.g. in [*de Bruijn 70a (A.2)*], [*van Benthem Jutting 77*]. For those who have not read anything about the project, this survey cannot pretend to give more than a vague idea of the languages. For getting a better idea, [de Bruijn 73b], [*van Daalen 73 (A.3)*] may be recommended; [*van Daalen 73 (A.3)*] gives a very precise definition of AUT-QE, one of the most prominent members of the family (see Sections 9, 13).

2. PURPOSE OF THE PROJECT AUTOMATH

The project was conceived in 1966; the first report was [de Bruijn 68b]. The idea was to develop a system of writing entire mathematical theories in such a precise fashion that verification of the correctness can be carried out by formal operations on the text. Here "formal" means: without "understanding"

*Reprinted from: Seldin, J.P. and Hindley, J.R., eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, p. 579–606, by courtesy of Academic Press Inc., Orlando.

the "meaning", and therefore it has to be possible to instruct computers how to check the correctness. Indeed, the fact that we do have computers will be one of the reasons why our generation has better chances than those who tried to have similar claims in the past, like Leibniz, Peano and Hilbert. Even if we do not actually *use* computers, they are there to set the standard of what is "formal" verification.

In the next three sections we discuss motivations for the project: checking, understanding and processing. The first two motives seem to favour the choice of a system of a very general nature, not necessarily tied to today's ideas of formalizing mathematics in terms of classical logic and set theory.

3. CHECKING

Most mathematicians can very well check themselves what they read and write. Nevertheless only a small portion of mathematical literature is absolutely flawless. Moreover, human checking seems to be a social affair too: mathematicians put trust in something since they think or know that other mathematicians have checked it.

Very meticulous checking is definitely unpopular. The thing we have in mind puts quite a burden on those who write the mathematics to be checked. They have to justify every little step extensively. It is only after this that a computer can do the final checking and guarantee the correctness.

We mention two cases where checking may be important. The first one is for things which are very hard and condensed, and where there is little intuitive or experimental support. The second one is for long and tedious proofs which form very long chains of very elementary steps. Such things may occur in combinatorial arguments, but, more important, in the large amount of work that has to be done to check the semantic correctness of large computer programs or machine designs.

Checking may actually be carried out in man-machine cooperation. This may also mean that, at least temporarily, parts of the checking may be omitted if they refer to things we are absolutely sure of.

Many errors in mathematics are made at the interfaces between theories. Therefore, we want to do the checking in a system that embraces all the theories involved. For example, if we want to check that the regular 17-gon can be constructed by ruler and compass, we have to be able to formulate the rules of geometric constructions into our system.

4. UNDERSTANDING

Formal systems help us to understand mathematics already by the mere fact that they force us to subdivide mathematical discussion into

(i) language,

(ii) metalanguage, and

(iii) interpretation.

The role of the latter is often underestimated. Those who say that mathematics *is* set theory, usually disregard the fact that they handle an extensive system of interpretation which is almost completely intuitive. Quite often it is just the interpretation that means "understanding" mathematics. Therefore we want a system that checks as much as possible of what we can actually say. (This is as far as we can go: we cannot expect a machine or a person to check what is in the back of our minds.) Our system should check a kind of language that comes as close as possible to what we write in ordinary mathematics.

If we want to understand mathematics we also have to get insight into the roles of axioms, definitions, proofs, theorems. We cannot expect to get such an insight from a basic theory that has been built up itself with axioms, definitions, proofs and theorems. It is much better to have a foundation that is nothing but a set of rules for manipulating language. On such a foundation we can build logic and mathematics, possibly with the use of axioms. There is nothing against axioms, but we should be free to accept them or to reject them. Axioms should not be tied to the fundamentals of our system.

Another thing that a good language may help us to understand is the structure or the complexity of an argument. The text may reveal analogies in the structure of arguments, and classification of their inherent difficulty. As to the classification of difficulty we mention that a very useful borderline between "elementary" and "higher" mathematics is that elementary mathematics is the part of mathematics that can be expressed without lambda calculus. In other words: "elementary" is what can be said in PAL (see Section 11).

5. PROCESSING

The fact that a machine can read, check and store the mathematics we produce, can have several advantages. One of these is that we can be absolutely sure that two mathematicians use the same theorem with exactly the same conditions. But a machine can also process its contents for answering questions.

Examples:

(i) produce a glossary of a text,

(ii) find out in a given argument whether a given axiom does or does not play a (direct or indirect) role,

(iii) print all notions and arguments that are needed to understand a given theorem, omit everything that is irrelevant to it.

6. WHAT KIND OF MATHEMATICS CAN WE DO?

The Automath system is like a big restaurant that serves all sorts of food: vegetarian, kosher, or anything else the customer wants. The languages are not tied to any logical system: hardly any logic has been built in. Admittedly there are basic notions of functionality and typing, but these need not be used the way they seem to be intended for. Those who want to say that a function is a subset in a certain cartesian product, can say it in Automath, but the restaurant also caters for those who want to describe mathematical functions by means of the functionality available in the language itself. Those who reject the axiom of choice or the excluded middle can use the system, as well as adepts in "New Math" and those who see "truth" as a matter of checking zeros and ones in truth tables.

Nevertheless some customers are better served than others. The best-served are those who try to keep close to the way mathematicians actually talk and think. They can use the types for doing typed set theory, the context structure to represent their ordinary way of reasoning (natural deduction), and the built-in functionality for describing their functions.

For typed set theory and natural deduction in relation to Automath, see [*de Bruijn 75a (F.1)*], [Nederpelt 77]. Formal Zermelo-Fränkel set theory was written in AUT-68 (cf. Section 9) by [van Daalen 70]. For a large piece of mathematics described in the "natural style", we refer to the Landau translation (see Sections 20 and 25).

7. BOOKS AND CONTEXTS

We write our mathematics in books, consisting of sequences of lines. Each line is written in some context.

We use the word "context" in a restricted sense. At each point of a mathematical discussion we can consider:

(i) The set of assumptions which are considered to be valid at that point.

(ii) The set of variables which are "alive" at that point.

(iii) The set of all notions that have been developed previously (either by definitions or by taking them as primitives).

Many people will say that the context is (i) + (iii), and disregard (ii) (their idea is that there is an infinite pool of variables which are always available).

We shall use the word context differently, taking it to be described by (i) + (ii). There is no reason for us to specify (iii), since it follows from the given order of the lines in the book. This is not true for (i): assumptions can be both introduced and discarded. And as to (ii): our point to take this as part of the context, is the fact that the variables will be typed. These types may be expressed by means of "older" variables but their construction may also depend on the fact that the assumptions of the context are valid (i.e. the types may be defined by expressions containing things that were defined only under these assumptions).

Similarly, the assumptions may be expressed in terms of variables belonging to the context. In this report assumptions and variables play the same role in the context. They can appear in any order. Let us give an informal example of a context:

> "Let n be a natural number. Let P be a point of R_n.
> Let Q be a point of R_n. Assume $d(P,Q) > n$."

This context contains three variables n, P, Q and one assumption. We say that this context has length 4. Things of the kind (iii) are "natural number", "point of R_n", "d", "$>$".

In a mathematics book we can indicate the context of every line. There is a special kind of lines that serve to define new contexts (these lines are called *block openers*). Examples: "Let n be a natural number". "Assume $d(P,Q) > n$". Block openers are placed in a context too.

A context can be seen as a sequence of block openers, arranged in the order in which they appear in the book. If these context lines are labeled $A_1, ..., A_n$, then the context of A_n is $A_1, ..., A_{n-1}$. Therefore the context $A_1, ..., A_n$ is adequately described by mentioning A_n only: looking up line A_n in the book will reveal A_{n-1}, etc.

The phrase "block opener" suggests the usual situation that assumptions are taken to be valid during a sequence of consecutive lines, and that validity regions of assumptions are nested intervals. These things will not be generally assumed however. A context can shrink for a while, and be picked up later.

8. DEFINITIONAL LINES AND PN-LINES

What kind of material can be written in a context (apart from block openers that extend the context)? It will turn out that we can get away with two things: definitional lines and PN-lines. In the first case we have a new identifier (symbol or word), and an expression (in terms of old identifiers and material from the context); the line is interpreted as the definition of the new identifier. In a PN-line, however, no expression is given, but the symbol PN is written instead. The interpretation is that the identifier is introduced as a primitive symbol. In Section 14 it will be explained how some of the definitional lines can be interpreted as theorems with proofs and some of the PN-lines as axioms.

9. THE LANGUAGE FAMILY

As basic language we take SEMIPAL. It is not able to handle mathematics, but just intended to give a record of how things are expressed in terms of others. The contexts in SEMIPAL are sequences of untyped variables. Apart from the block openers there are definitional lines and PN-lines. The expressions are composed of identifiers and variables. If the context is $x_1, ..., x_n$, the new identifier is p, then the line is written as something like

$$x_1, ..., x_n \; * \; p \; := \; f(g(x_1, a), h(x_1)) \; . \tag{9.1}$$

On the right we have an example of an expression. In order to explain what we intend with this line, it is better to write $p(x_1, ..., x_n)$ instead of p; the interpretation is that p is introduced as a function of n variables. The expression on the right is assumed to be correct, i.e.

(i) each non-variable identifier has been introduced previously in the book, with a context length equal to the number of subexpressions it has in (9.1),

(ii) the variables occurring in (9.1) all belong to the context $x_1, ..., x_n$.

SEMIPAL can be extended in two ways:

(i) By admitting lambda expressions (λ-SEMIPAL).

(ii) By attaching a type to every expression, taken from a fixed finite set of types. Let us call this PAL-FT (PAL with fixed types).

We can go beyond (ii):

(iii) By admitting the introduction of type variables and of primitive types. This will be called PAL ("Primitive Automath Language").

The combination of PAL and λ-SEMIPAL leads to AUT-68 (for a long time this was called Automath), and, a little beyond it, AUT-QE. Let us write $A \leq B$ if every correct book in language A is also correct in language B. Then we have PAL $\leq$ AUT-68 $\leq$ AUT-QE.

A different extension of PAL is J. Zucker's AUT-II (see Section 22).

The language AUT-SL (single line AUT, see [*de Bruijn 71 (B.2)*], [*Nederpelt 73 (C.3)*]) has been created mainly in order to get a streamlined language theory. It is a very general higher-order language, obtained by giving up all restrictions on abstraction, and admitting all numbers 0,1,2,... as degrees (see Section 11). Once this has been done, we can write PN's as block openers (cf. Section 16), eliminate all definitional lines, and thus obtain a complete book in the form of a single line.

10. ABBREVIATION SYSTEM

In SEMIPAL we have a simple abbreviation system that can be maintained throughout the language family. If p was introduced by (9.1), say, then in later expressions p is allowed to have fewer than n subexpressions. The missing subexpressions are just supplied by adding $x_1, x_2, \ldots$ on the left. For example: if $E_3, E_4, \ldots, E_n$ are expressions, then $p(E_3, \ldots, E_n)$ is an abbreviation for $p(x_1, x_2, E_3, \ldots, E_n)$. (So $p(E_3, \ldots, E_n)$ can only be used in a context containing the first two variables of the context of (9.1).)

Quite a different kind of abbreviation (again for all languages of the family), lies in the *paragraph system* (for a description see [*van Benthem Jutting 77*]). It has the practical advantage that names for identifiers (e.g. common letters like $x, a, \ldots$) can be used over and over again. The book is divided into sections, subsections, sub-subsections,... (all called paragraphs). If we mention an identifier we mean the one that was introduced in the smallest surrounding paragraph; if we want to refer to a different identifier with the same name, we have to mention its paragraph number.

11. TYPING AND DEGREES

We begin with a language with fixed types. Let us call it PAL-FT. We start from SEMIPAL, and we attach a type (taken from the given set) to every variable, to every identifier and to every expression. The rules are obvious: if we form an expression by substituting expressions $E_1, \ldots, E_n$ for $x_1, \ldots, x_n$ in $p(x_1, \ldots, x_n)$, then for each i the type of E_i should equal the one of x_i; and $p(E_1, \ldots, E_n)$ gets the same type as p. The type can be written at the end of each line of the book (including block openers). As a separation mark we can use the semicolon (we also write $p : \tau$ in the metalanguage in order to say that p has the type τ).

Let us pass to PAL. We introduce a new symbol **type**, and say that τ : **type** for every type τ we had thus far. Let us admit this new kind of typing for block openers as well as for PN-lines. Then by obvious extensions of our rules, we can get the new types in the definitional lines too. We do not need the collection of fixed types anymore: the same effect can be obtained with PN-lines "τ := PN : **type**".

Since PN-lines can be written inside a context, we can get big expressions typed by **type** (i.e. we get types depending on a number of parameters).

Let us say that **type** is an expression of degree 1; if E : **type** we say that E has degree 2; if $F : E$ and E : **type** we say that F has degree 3.

In the languages mentioned in Section 9 the degrees are restricted to 1,2,3. There would not be any harm in admitting higher degrees, but the description of present-day mathematics does not seem to require more than three degrees. There is a suggestion of using degree 4 in [*de Bruijn 74b (B.3)*], but what is done with it might also be done with lower degrees by slight modifications of the language.

The typing rule of PAL-FT is to be modified in PAL: the type of $p(E_1, ..., E_n)$ is to be what we get if in the type of $p(x_1, ..., x_n)$ we substitute E_1 for $x_1, ..., E_n$ for x_n. And we require that the type of E_i is "definitionally equal" (see Section 18) to the one we get by that same substitution in the type of x_i (the latter type does not contain $x_i, x_{i+1}, ..., x_n$).

12. ADDING THE LAMBDA CALCULUS

In Section 9 we announced λ-SEMIPAL as what we get from SEMIPAL by admitting λ-expressions as expressions. (This language has never been used or studied in the project; it is only mentioned here as a resting-point in the discussion.) If E is an expression containing the variable x, then $\lambda_x E$ is an expression in which x is no longer a variable but a dummy. The passage from E to $\lambda_x E$ is called *abstraction.* The interpretation is that $\lambda_x E$ is a function, which at any point p has as its value the expression we get if in E we replace x by p (the result of this substitution is written in the metalanguage as $E\,[x := p]$).

The counterpart of abstraction is called *application.* We write $\langle p \rangle\, f$ for the thing that is interpreted as the value of the function f at the point p. (The usual way of writing fp or $f(p)$ is inconvenient since abstraction is written on the left, and it happens so often that abstractions and applications are tied together in pairs.

A crucial role in the metalanguage is played by *β-reduction.* This means reducing $\langle p\rangle\,\lambda_x E$ to $E\,[x := p]$, in accordance with the interpretation. Less important is *η-reduction,* reducing $\lambda_x\,\langle x\rangle\,E$ to E in cases where E does not contain x.

In λ-SEMIPAL we have two different ways to describe the relation between a function f, a value p of the variable, and the value of the function at that point. One way is the application $\langle p\rangle\,f$, the other one is by means of what we shall call *instantiation.* If f is an identifier introduced in the context x (either by a definitional line or by a PN-line or by a block opener) then we can use the expression $f(p)$ in later lines. This feature of the language has disadvantages (two ways of writing, with the same interpretation) and hardly any advantage: instantiation does not do what application cannot do. This will be different in typed languages: the scopes of instantiation and application overlap, but none of the two scopes is contained in the other.

13. ADDING TYPED LAMBDA CALCULUS TO PAL

We first say that the word "typed" in the title does not refer to fixed types like in PAL-FT of Section 11. We shall admit type variables, and lambda expressions as types. Therefore we get beyond what is usually called typed lambda calculus.

The typed lambda expressions we want to add to PAL are of the form $\lambda_{x:A}B$. The B may contain x as a variable, and it has to be a legitimate expression under the assumption that the type of x is A. In the metalanguage we speak of "abstraction over A" or "abstraction of B over A".

The subscripted notation $\lambda_{u:U}$ is hard to print in the many cases where U is an expression containing further λ's. Therefore we always write $[u : U]$ instead of $\lambda_{u:U}$.

There are various possibilities to play the game. For a survey we refer to [de Bruijn 74a]. In particular we have to decide what degrees for A and B we admit. Both in AUT-68 and AUT-QE we admit abstraction over A's of degree 2 only. In AUT-68 the abstracted expression B can have degree 2 or 3, in AUT-QE B can have degree 1, 2 or 3. The typing rule for λ-expressions in AUT-QE is roughly this: if in the context γ extended by $x : A$ we have $B(x) : C(x)$, then in the context γ we have $[x : A]\,B(x) : [x : A]\,C(x)$. In AUT-68 this is different if B has degree 2. If $B(x) : \texttt{type}$ then AUT-68 obtains $[x : A]\,B(x) : \texttt{type}$.

In AUT-QE the "quasi-expressions" (like $[x : A]\,\texttt{type}$) seem strange, but once one gets accustomed to them they turn out to be quite natural and enjoyable. They allow applications $\langle a\rangle\,f$ if we know $f : P$, $P : [x : A]\,\texttt{type}$ and $a : A$.

There is a rule in AUT-QE that increases the power of the language. The rule is called *type inclusion.* If we have a typing like $T : [u : U]\,[v : V]\,\texttt{type}$ we say that the typing $T : [u : U]\,\texttt{type}$ and $T : \texttt{type}$ are also acceptable (acceptable in the sense of the rules for instantiation and application). Expressed superficially: everything we say for arbitrary types can be used for function types too.

Actually we can take three decisions about type inclusion. It can be forbidden (like in AUT-SL), allowed (like in AUT-QE) or prescribed (like in AUT-68). Prescribing type inclusion means that the abstractions in front of `type` have to be skipped.

In AUT-68 typings are unique in the following sense. If in some context both $A : B_1$ and $A : B_2$ are correct, then B_1 and B_2 turn out to be definitionally equal (see Section 18). In AUT-QE this holds with the exception of type inclusion. But this is just a matter of phrasing the language definition. We can also say that typing is unique but that the typing rule is liberalized (cf. "mock typing" in [de Bruijn 74a]).

If $A : \texttt{type}$ and $B : \texttt{type}$ we are able to say "let f be a mapping of A to B" by means of a block opener "$f : [x : A]\,B$". This shows that in the typed language the lambda calculus can do what instantiation cannot do (cf. Section 12). On the other hand, by instantiation we are able to handle block openers like "$A : \texttt{type}$", and the functional relationships expressed in this context cannot be expressed by abstraction, at least not in languages (like AUT-68, AUT-QE) that forbid abstraction over expressions of degree 1 (e.g. over `type`).

14. USE OF TYPING FOR REASONING

The fact that PAL and its descendants AUT-68 and AUT-QE can be used for mathematical reasoning depends on the idea of *propositions as types.* Roughly it means that if p is a proof for a proposition, we write it as a typing $p : P$. This principle goes back to [Curry and Feys 58], and was elaborated by [Howard 80], [Prawitz 71], [Girard 72], [Martin-Löf 75a]. Completely independent of these developments it appeared in [de Bruijn 68b], [*de Bruijn 70a (A.2)*].

Treating propositions as types is definitely uncommon for the ordinary mathematician, yet it is very close to what he actually does. We shall try to explain this presently.

Assume that our book contains the following theorem (described informally), for some given functions φ, ψ:

Theorem 1. *Let x be a real number. Assume $\psi(x) > 1$. Let n be an integer. Assume $\varphi(x) > x^n$. Then $\psi(x) > n$.* □

We want to apply this later, with $x = q$, $n = 5$, and want to conclude $\psi(q) > 5$.

We have to convince ourselves that the conditions are satisfied. To this end we write a proof for $\psi(q) > 1$ and label this result as (1). And we write a proof for $\varphi(q) > q^5$ and label that result as (2). Now we claim to apply the theorem, providing in this order q, (1), 5, (2). So the (1) and (2) are treated on a par with the names (of "objects") q and 5. Is (1) to be considered as a name for the proposition $\psi(q) > 1$? No, the application of the theorem is not legitimate because of the existence of the proposition $\psi(q) > 1$, but because of its being proved. So consider the reference (1) as a reference to a proof of $\psi(q) > 1$. Let us try to explain our application to a machine that knows Theorem 1. The machine wants to check

(i) that q is a real number,

(ii) that (1) is a statement that $\psi(q) > 1$ has been proved,

(iii) that 5 is an integer,

(iv) that (2) refers to a proof of $\varphi(q) > q^5$.

We only need to change a few words in order to get: q is a real number, (1) is a proof of $\psi(q) > 1$, 5 is an integer, (2) is a proof of $\varphi(q) > q^5$. Altogether, we have a proof of $\psi(q) > 5$.

The parallelism between proofs and "ordinary" mathematical objects gets even stronger if we realize that many objects are defined conditionally only. If we define a function f for x real, $x > 1$ then the use of the value of the function at a point requires

(i) that point (a real number),

(ii) a proof that the real number is > 1.

Now the value of the function is an object, and it depends on an object and a proof. So proofs may depend on objects and objects may depend on proofs. One might say that we have been confusing "proofs" with "references to proofs" or "names of proofs". But in informal talk we make the same switchings from "objects" to "names of objects". There is not much of a point in arguing whether proofs are as real or more real than objects. Quoting Wittgenstein's "Don't ask for the meaning, ask for the use", we must say that as far as the use is concerned, the parallelism is complete.

In the above example, the proof of $\psi(q) > 5$ is a single-step proof. In PAL it is expressed in a line

$$\text{Theorem 2} := \text{Theorem 1}(q,\ (1),\ 5,\ (2))\ :\ P$$

where P in some way represents the proposition $\psi(q) > 5$ (or rather the type of proofs of that proposition). The term "single-step proof" means that we only have to quote. It would become a multi-step proof if (1) was not available directly in the book, but (1) had to be constructed on the spot, again by substituting things in the name of the proof of a theorem, like "lemma $3(q,q)$" in

$$\text{Theorem } 1(q, \text{ lemma } 3(q,q),\ 5,\ (2))\ :\ P\ .$$

In this way arguments of several steps can be condensed into a single line.

Let us illustrate the principle "propositions as types" by how it works for implications. Let p and q be propositions. Having a proof of the implication $p \to q$ can be interpreted as this: we have a procedure by which we are able to give a proof of q for every customer who might present us a proof of p. That is, our procedure is a function that maps the set of all proofs of p into proofs of q. Using our terminology of context, we can say that in the context "x : proof(p)" (representing "let x be a proof of p") we can write a line

$$f\ :=\ \ldots\ :\ \text{proof}(q)\ .$$

By the abstraction rule of AUT-68 or AUT-QE we get, outside the x-context (see Section 13 for the notation),

$$[t\ :\ \text{proof}(p)]\,f(t)\ :\ [t\ :\ \text{proof}(p)]\,\text{proof}(q)\ .$$

Hence $[t\ :\ \text{proof}(p)]\,\text{proof}(q)$ acts as the proof type of the implication.

15. USING TWO EXPRESSIONS OF DEGREE 1

For various reasons it is attractive to introduce a symbol `prop` of degree 1 that behaves exactly like `type`, but with different interpretation. If $A\ :\ B$, B : `type` then A is the name of an object of type B, and if $C\ :\ D$, D : `prop` then C is the name of a proof for the proposition expressed by the proof type D.

One reason to make the distinction between `type` and `prop` is to give an easier insight into the interpretations, but there are also more essential reasons for making the difference. One of the forms of the logical double negation axiom, written by means of "`prop`", turns into the axiom about Hilbert's ε-operator if we replace `prop` by `type`. So if we want to do classical logic and do not want to accept the axiom of choice, we need some distinction. It should be mentioned, however, that introduction of `prop` is not the only way out of this difficulty. (Another way is to create a primitive type called "bool" (for boolean) and for every boolean b a primitive type "proof type of b".)

Another suggestion to profit by treating type and prop differently, is "proof irrelevance" (Section 24).

We can now give a survey of the various kinds of lines involving prop. First, block openers "x : prop" introduce propositional variables. PN lines "p := PN : prop" introduce primitive propositions. A definitional line "b := ... : prop" introduces an abbreviation for a more complex expression representing a proposition.

Next we take some P with P : prop. This P is interpreted as the proof type of a proposition. Now the block opener $x : P$ states the proposition as an assumption. The PN-line u := PN : P is interpreted as stating the proposition as an axiom. The definitional line $v := E : P$ states the proposition as theorem. The expression E represents the proof, and v is a name for the proof. The theorems themselves do not get names. In order to quote a theorem it suffices to quote a name for the proof.

Contexts are sequences of block openers like

$$x_1 : A_1, ..., x_n : A_n .$$

At the places where A_j : prop the interpretation is that x_j is the name of the assumption, at places where A_j : type the x_j is a variable. And, of course, there can be places where A_j = type or A_j = prop.

Especially in AUT-QE it is attractive to talk "prop-style", i.e. to suppress all propositions and talk about their proof types only. It turns out that there is hardly ever a necessity to talk about the propositions any more (talking about propositions is called "bool-style"). The example at the end of Section 14 shows how this works: we can just *define* the proof type of the implication associated with the proof types P and Q by $[t : P]\,Q$.

The more often one does this kind of thing, the easier one forgets the original use of the word "proposition". This may explain why the Automath workers began to say prop instead of proof type. A consequence is that if P : prop they do not pronounce $p : P$ as "p is a P" but as "p proves P".

16. AXIOMS VS. ASSUMPTIONS

If we have a PN-line in an empty context there is no harm in replacing it by an assumption. The name of the assumption will be a part of every context in the sequel. Taking it as an assumption gives more flexibility, since axioms are things we can never get rid of (unless we start a new book) and assumptions can be discarded if we wish.

If a PN-line is written in a non-empty context we can sometimes, but not always, replace it by an equivalent axiom in the empty context (and next replace it by an assumption). Whether this is possible depends on the degrees involved in the context as well as on the degree of the type of the PN-line, both in connection with the abstraction rules of the language. In AUT-SL, the most liberal language of the family, all PN's can be eliminated this way.

17. DERIVATION RULES

In the Automath family there is no essential difference between logic and mathematics. Logical connectives can be taken as PN's or as defined notions, inference rules can be taken as axioms or as derived rules, and later applications of such rules have the same form as applications of mathematical theorems.

As an example we present the double negation law. Somehow we have an expression *CON* with *CON*:`prop`. It has the following interpretation: if in some context we have an expression p with p : *CON*, then "we have a contradiction" (one can even say that p *is* a contradiction). In the context P : `prop` we next define *NON*(P) (by means of a definitional line) as $[x : P]$ *CON*. The "double negation law" can now be written as follows:

$$[P : \texttt{prop}]\,[y : NON(NON(P))] * dbng := \mathrm{PN} : P\,.$$

To the left of the asterisk the context is indicated: "let P be a `prop`, let y be a proof of the double negation". In this context we postulate the truth of P. The identifier "*dbng*" is chosen as the name of the law.

18. TWO KINDS OF EQUALITY

There is (already in SEMIPAL) a notion of definitional equality between expressions. The notion plays a central role in language theory. In typed languages it is essential already in the language definition (see the end of Section 11). Definitional equality is generated by δ-reductions (δ-reduction means elimination of some previously defined identifier, replacing it by its definition given in the definitional line) and the β- and η-reductions of the lambda calculus.

In our languages no facilities have been provided for talking in the book about definitional equality. It is hardly necessary, for if A and B are definitionally equal then at every place in the book A may be replaced by B without any argumentation. The kind of equality mathematicians do talk about is what we call *book equality*. It may be introduced by means of a PN (but there are also possibilities to *define* book equality), and its basic properties can be covered by axioms or theorems.

19. LANGUAGE THEORY

Language theory is about reductions (the δ-, β- and η-reductions mentioned in Section 18), normal forms (i.e. expressions which do not admit reductions) and about the relation between correct expressions and their types ("correct" means: acceptable in the book). Important parts of the language theory were obtained in [van Benthem Jutting 71a], [*van Daalen 73 (A.3)*], [*Nederpelt 73 (C.3)*], [*de Vrijer 75 (C.4)*]. The forthcoming Ph.D. thesis [*van Daalen 80*] will cover all aspects of the language theory at least for AUT-68, AUT-QE and AUT-SL. The essential results are (in a rough formulation)

(i) The Church-Rosser theorem: If A and B are definitionally equivalent then there is an expression C such that both A and B can be reduced to C by sequences of reductions.

(ii) The normal form theorem: For every A there is a normal form N to which A can be reduced by a sequence of reductions; N is uniquely determined.

(iii) The strong normal form theorem: Every reduction sequence terminates (and for every A there is an upper bound to the length of the reduction sequences starting at A).

(iv) The closure theorem: If A is correct and if A reduces to B then B is correct.

We note that ii) is not true for untyped lambda calculus. It is true, however, for the untyped language SEMIPAL (which has no lambdas).

20. VERIFICATION

One of the most important things in the project is that we expect machines to check the correctness of what humans have written. This would be an easy programming job if the language would require of the writer that every little application of the rules of the language should be indicated in the text. But this is out of the question: from experience we know that it would require texts which are hundreds of times longer than they are in our present system. We expect the machine to do much of the checking on its own initiative, not necessarily in the same way the textwriter might have had in mind.

The machine has to find out whether there is a sequence of applications of the language rules that motivates the correctness of a line of the book, once all previous lines have been checked. The results of language theory show (at least for SEMIPAL, PAL, AUT-68, AUT-QE, AUT-SL) that this is automatically decidable. Definitional equivalence of two expressions can be established by reducing both to their normal form and checking whether these are the same. But already in short books this may turn out to give a prohibitive amount of work (in particular it will duplicate much of the work in checking previous lines). What we really want is a good *strategy* by which the machine can try to find a shorter way from one expression to the other, about as short as what may have been in the writer's mind.

The computer programs whose execution effectuates the verification of a book, are called *verifiers* or *checkers.* For AUT-68 and AUT-QE the verifiers operate satisfactorily. The checkings can be done on-line from a teleprinter. In some cases where the program's strategy seems to run into very much work, the machine may ask whether the writer really wants it. In most cases it turns out that the writer has made a mistake. It would not be sensible to require that the machine *proves* that a line is incorrect: such a proof might require evaluation of normal forms. Therefore, it is better to let the machine report if it has a serious difficulty. And on some rare occasions we may let the machine ask for a hint in what direction to search. Sometimes it may help the machine if we write a few extra lines in the book, just as if we are explaining mathematics to human readers. In general, if we condense two lines into one, then checking the condensed line may require more work than checking the separate lines one by one.

Most of the work on verifiers was done by I. Zandleven (cf. [*Zandleven 73 (E.1)*]) in the years 1971–1976. Later this work was continued by A. Kornaat and L.S. van Benthem Jutting. A very large part of the effort is just caused by the limitations of today's computer technology. The amount of information involved in handling moderate amounts of mathematics is so big that it has to be distributed efficiently over the various kinds of fast and slow memory, and checking a single line may require consultation of many remote parts of the book. The paragraph system (see Section 10) plays a role in coping with these difficulties.

In handling substitution in lambda calculus it is often necessary to re-name dummies in substitution operations, in order to avoid "name clashes". In order to simplify this, namefree lambda calculus was developed by [*de Bruijn 72b (C.2)*], [de Bruijn 78a], where references to dummies are not indicated by name but by reference depth. This system lies at the root of today's verifiers.

As it was said before, the problem of how to handle large amounts of mathematics requires considerable effort in the design of the verifiers, but the matter of strategies is more essential. It is, of course, closely related to language theory. The closure theorem (Section 19) is important: it saves much work, e.g. it saves checking types when doing β-reduction.

The essential difficulty of verification is also the essential difficulty of language theory. It is the fact that definitionally equal expressions are connected by chains $A_1, A_2, ..., A_n$ in which the reductions go either way: sometimes A_i reduces to A_{i+1}, sometimes A_{i+1} reduces to A_i.

[*van Benthem Jutting 77*] gives some details about experiences with the checking of a relatively large text (viz. the translation of Landau's "Grundlagen", [Landau 30]). The coded version ([van Benthem Jutting 76]) consists of about 5.10^7 bits. This may seem very large (maybe 10 to 50 times as large as a direct encoding of the words and symbols Landau wrote himself), but it is still of the order of what a single cassette tape can contain.

21. AUTOMATIC THEOREM PROVING

Automatic theorem proving is a very hard subject. In order to be efficient it certainly requires clever adaptation to the kind of problems it is applied to. Therefore it is very questionable whether it would profit much from Automath, with its claims for generality and adaptivity to human reasoning. Admittedly, our verifiers do automatic searching, and may establish definitional equalities the writer has not bothered to see through, but this is not the level of what is usually called automatic theorem proving.

Nevertheless one may think of building "attachments" to the verifier which find proofs of little gaps the writer might like to leave. This might be done completely outside the system (e.g. by consulting the computer's arithmetic unit or by checking tautologies by inspection of cases), but it can also be conceived that the machine, after finding its proof, writes it in Automath and checks it by its own verifier. An attachment of the latter type was built (as a student's exercise) by R.M.A. Wieringa. Given natural numbers p, q, r with $pq = r$, where p, q, r are presented in the binary number system, his program produces an Automath text proving $pq = r$. The number of lines is of the order of the number of digits we write down with ordinary pencil-and-paper multiplication.

Attachments of the first kind, operating outside the system, can of course work very much faster. At least some of them will be very profitable, but the Automath group never worked in this direction. They rather did what others don't than what others do very efficiently already.

22. FURTHER LANGUAGE EXTENSIONS

There is a number of things that mathematicians find so self-evident that they do not see them as part of the structure of axioms, definitions and theorems, but more as a part of their language. This is deceptive, of course (after all we seem to do a lot of mathematical work subconsciously), but nevertheless one can try to incorporate as much as possible in the language definition. To quote a few unrelated things: pairs, strings, set theoretical operations, equality, commutativity and associativity, mathematical induction. One might say that in AUT-68 and AUT-QE only two things have been implemented: functional relationship and typing. All the rest is left to the book-writers.

We never found much use for building mathematical induction into the language definition (it is done in some other constructive systems). The reason is that in our system we have books to write in, and for a thing like induction it is as easy to quote the rule from a book as to apply a language facility. But for some of the other subjects mentioned above, the use of language facilities would be very much shorter than quoting from the book.

Every extension may seriously complicate both language theory and verifier. It is not clear how far one should go. J. Zucker devised AUT-Π ([*Zucker 77 (A.4)*]) as a relatively mild adaption and extension of AUT-68. It is much easier to write than AUT-68. Zucker wrote an extensive manuscript "Real Analysis" directly in AUT-Π (it is not a translation of something that was written first in ordinary language on scrap paper). A few chapters were written by A. Kornaat, who also produced some harder material in AUT-Π, viz. the proofs of the equivalence of various forms of the axiom of choice.

AUT-Π uses some proper extensions (like facilities for handling pairs) and a number of things which are more in the line of fast notation. Much of this belongs to a system called AUT-SYNT (partly developed by I. Zandleven) which has facilities for operations on syntactic variables, strings, and telescopes (a telescope is a string of block openers with types, like $[x_1 : A_1] \dots [x_n : A_n]$; the name comes from hand telescopes with tubes fitting into each other).

The work on language theory and verifier of these languages is unfinished.

One way to look at AUT-SYNT is that it is just an auxiliary language (like in [de Bruijn 72a]) that helps us to prepare an input text in a language like AUT-68 or AUT-QE, where language theory and verifier are on pretty safe grounds. It is likely that on the long run AUT-68 provided with AUT-SYNT input facilities will not be less adequate than some of the fancier languages, at least for classical mathematics.

23. IS THERE A NEED FOR HIGHER ORDER LANGUAGE?

As it was said before (Section 13), AUT-68 is a first order language since there is no abstraction over expressions of degree 1. Yet this does not seem to be a serious limitation, since a few extra axioms in the book extend the power of the language. As an example, we mention how abstraction over `prop` can be mimicked. We start wity an axiom in the empty context "$bool :=$ PN: `type`", and from now on expressions b with $b\ :\ bool$ are interpreted as propositions. Next, in a context $[x : bool]$ we take the axiom "$proof :=$ PN: `prop`". (In older publications we wrote "*TRUE*" instead of "*proof*".) The effect is that for every proposition b the typing "$u : proof(b)$" will mean that u is a proof for b. Now we can mimic abstraction over `prop`. Instead of saying that $f(p)$ holds for all p : `prop`, we say that $f(proof(b))$ holds for all $b : bool$, and the abstraction is now over something of degree 2.

The reports [de Bruijn 76], [de Bruijn 77], [*de Bruijn 78c (B.4)*] give suggestions how slight extensions of AUT-68, and how AUT-QE-NTI (AUT-QE without type inclusion) can be used for mimicking stronger languages.

24. PROOF IRRELEVANCE

This is a feature we might add to our languages if we are interested in classical mathematics only. The classical mathematician would find it even hard to understand what its counterpart "proof relevance" is. We give an example. If x is a real number, then $P(x)$ stands for "proof of $x > 0$". Now we define "log" (the logarithm) in the context $[x : \text{real}]\ [y : P(x)]$, and if we want to talk about log 3 we have to write $\log(3, p)$, where p is some proof for $3 > 0$. Now the p is relevant, and we have some trouble in saying that $\log(3, p)$ does not depend on p. This can be done by means of the general axioms for book equality, with the effect that in this case $\log(3, p_1)$ and $\log(3, p_2)$ are book-equal if both p_1 and p_2 are proofs of $3 > 0$.

Some time and some annoyance can be saved if we extend the language by proclaiming that proofs of one and the same proposition are always definitionally equal. This extra rule was called "proof irrelevance" in [*de Bruijn 74b (B.3)*]. We of course do not want to have the similar feature for `type`.

25. MATHEMATICS PRODUCED IN AUTOMATH

As a test case for handling larger amounts of mathematics, [van Benthem Jutting 76] gave a line-by-line translation of Landau's "Grundlagen" ([Landau 30]) into AUT-QE. His experiences are reported in [*van Benthem Jutting 77*]. Landau's book was chosen because it gives material of different kinds in a very constant style of presentation: the steps do not get bigger towards the end of the book. It would of course have been much easier to rewrite Landau's book first, so as to make it easier to translate, but it was our aim to show that Automath can cope with any kind of mathematics, not just the mathematics especially designed for it.

Another substantial piece of work is Zucker's "Real Analysis" (mentioned in Section 22). And J.T. Udding writes (in AUT-QE) a new theory of real numbers based on an approach that avoids the repeated troublesome embeddings Landau had to go through [Udding 80].

Many smaller pieces of mathematics have been done by students. The experience is that in a period of 2 or 3 weeks a mathematics student (without any training in logic) is able to learn AUT-QE, produce a piece of text (possibly using basic material already known to the computer), punch it, have it checked via a teleprinter, correct it, and get a final AUT-QE version. For an account of how a piece of mathematics is translated in several stages see [van Benthem Jutting 73].

The easiest things to translate are very condensed and very abstract pieces of mathematics. (Example: the proofs of equivalence of various forms of the axiom of choice dit not become much longer than the original text.) Hard subjects are those where (subconscious) "experience" comes in, like in analysis and combinatorics.

A very important thing that can be concluded from all writing experiments is the *constancy of the loss factor*. The loss factor expresses what we loose in shortness when translating very meticulous "ordinary" mathematics into Automath. This factor may be quite big, something like 10 or 20, but it is constant: it does not increase if we go further in the book. It would not be too hard to push the constant factor down by efficient abbreviations.

26. WORK IN PROGRESS

Apart from things discussed before, we mention a few sub-projects which are studied now or will be studied in the near future.

(i) Programming language semantics. This may become an important customer for Automath. The idea is, to write in a single book: definition of a programming language and of its semantics, the logic and mathematics involved, particular programs, and proofs for their semantics. The ideal situation is this: a computer that has to execute a program, reads it directly from that book, thus avoiding every kind of interpretation. R.M.A. Wieringa is working on a system proposed in [de Bruijn 75b].

(ii) A far reaching extension of lambda calculus is presented in [de Bruijn 78a] and studied by [Wieringa 78]. In ordinary lambda calculus we can interpret substitution and β-reduction as replacing end-points of trees by branches of trees. The extension in [de Bruijn 78a] means that we can also break open some edge and paste a segment of a tree into it. These segments might represent strings or telescopes (see Section 22). This kind of lambda calculus can be expected to be helpful to simplify both language definitions and verifiers.

(iii) In the spirit of the work of the Automath project, the project WOT was started. WOT is a Dutch abbreviation and stands for the "mathematical vernacular", i.e. the strange mixture of words and formulas mathematicians use. The idea is to get to a purified form of WOT that can be used as a formal system for expressing mathematics. The foundations of mathematics have to become some kind of grammar for WOT. Thus far the only reports on WOT are in Dutch, and are used in the training of mathematics teachers.

ADDITIONAL NOTE

After this paper was finished, Professor Dana Scott pointed out that Section 14 should have mentioned H. Läuchli's work on the principle of propositions-as-types. Reference should have been made to his abstract "Intuitionistic propositional calculus and definably non-empty terms", J. Symb. Logic 30 (1965) p. 263, and to his paper [Läuchli 70].