Fundamenta Informaticae 185(4) : 313–336 (2022)
DOI 10.3233/FI-222112
IOS Press

Characteristics of de Bruijn's early proof checker Automath

Herman Geuvers[*]
Radboud University Nijmegen and
Eindhoven University of Technology
Nijmegen, The Netherlands
herman@cs.ru.nl

Rob Nederpelt
Eindhoven University of Technology
Nijmegen, The Netherlands

Abstract. The 'mathematical language' Automath, conceived by N.G. de Bruijn in 1968, was the first theorem prover actually working and was used for checking many specimina of mathematical content. Its goals and syntactic ideas inspired Th. Coquand and G. Huet to develop the calculus of constructions, CC ([1]), which was one of the first widely used interactive theorem provers and forms the basis for the widely used Coq system ([2]).

The original syntax of Automath ([3, 4]) is not easy to grasp. Yet, it is essentially based on a derivation system that is similar to the Calculus of Constructions ('CC'). The relation between the Automath syntax and CC has not yet been sufficiently described, although there are many references in the type theory community to Automath.

In this paper we focus on the backgrounds and on some uncommon aspects of the syntax of Automath. We expose the fundamental aspects of a 'generic' Automath system, encapsulating the most common versions of Automath. We present this generic Automath system in a modern syntactic frame. The obtained system makes use of λD, a direct extension of CC with definitions described in [5].

[*]Address of correspondence: H. Geuvers, Faculty of Science, Radboud University, PO Box 9010, 6500 GL Nijmegen, The Netherlands.

Received June 2021; accepted April 2022.

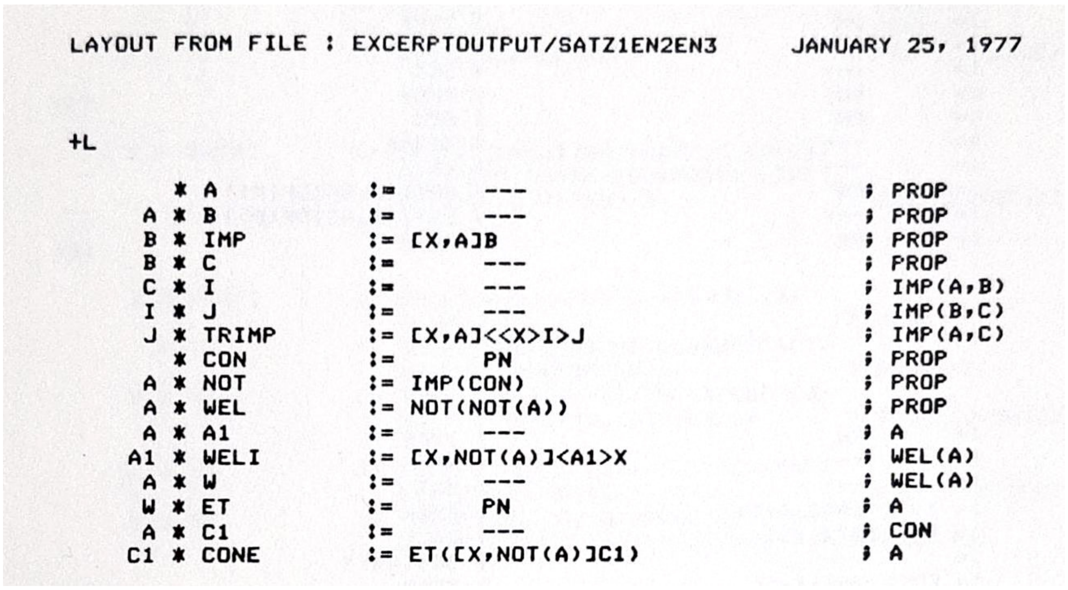
1. Introduction

Around 1967, N.G. de Bruijn started thinking about a formal checker for mathematical proofs. He defined several Automath languages, of which AUT68 and AUT-QE (see [6]) are the most used ones. They differ in 'strength' as to the abstraction facilities, but have a common structure. Below, we concentrate on the general aspects of the Automath languages, without differentiating between the different 'dialects'. Therefore we speak about 'Automath' only and discuss its structure in a *generic* manner.

The purpose of this paper is to give an impression of what Automath is and discuss in passing what de Bruijn intended with this mathematical language. In Section 2, we describe what the features of Automath are, as compared to other modern type theories and proof assistants based on type theory. In particular we focus on the 'book format' of Automath, with the idea that a formal mathematical text is built up incrementally, line by line, where each new line has to be correct in terms of the already existing part of the book, and thereby one creates a correct Automath book. In Section 3 we define formally what this correctness means in type-theoretic terms. In Section 4 we discuss de Bruijn's basic ideas about Automath and we summarise how Automath relates to modern type theories.

2. The Automath format

To give an idea of the original syntax of Automath, we reproduce a short fragment from the beginning of the formalization of Landau's Grundlagen ([7]), written by L.S. van Benthem Jutting ([8]) in Figure 1. In Section 2.6 we give a translation of this text in (more or less) common mathematical phrasings.



```
LAYOUT FROM FILE : EXCERPTOUTPUT/SATZ1EN2EN3     JANUARY 25, 1977


+L

       * A       :=      ---                 ; PROP
     A * B       :=      ---                 ; PROP
     B * IMP     := [X,A]B                   ; PROP
     B * C       :=      ---                 ; PROP
     C * I       :=      ---                 ; IMP(A,B)
     I * J       :=      ---                 ; IMP(B,C)
     J * TRIMP   := [X,A]<<X>I>J             ; IMP(A,C)
       * CON     :=      PN                  ; PROP
     A * NOT     := IMP(CON)                 ; PROP
     A * WEL     := NOT(NOT(A))              ; PROP
     A * A1      :=      ---                 ; A
    A1 * WELI    := [X,NOT(A)]<A1>X          ; WEL(A)
     A * W       :=      ---                 ; WEL(A)
     W * ET      :=      PN                  ; A
     A * C1      :=      ---                 ; CON
    C1 * CONE    := ET([X,NOT(A)]C1)         ; A
```

Figure 1. An example of an Automath text

As can be seen from this example, an Automath text consists of consecutive lines, each containing four expressions. These lines can have either of the three forms (i), (ii) or (iii) of Figure 2.

	indicator		identifier		content		category	
(i)	ϵ/x	$*$	z	$:=$	—	$:$	A	assumption line
(ii)	ϵ/x	$*$	c	$:=$	M	$:$	A	definition line
(iii)	ϵ/x	$*$	c	$:=$	PN	$:$	A	primitive line

Figure 2. The three species of Automath lines

In this scheme, ϵ (denoting the empty expression) is a special symbol, x and z represent variables, c denotes a constant and M and A are expressions; also PN acts as a special symbol. The symbols $*$, $:=$ and $:$ between the expressions are separators. [1]

The *first* entry of all Automath-lines, the *indicator* or *context indicator*, is the special symbol ϵ or a variable (here with the name x); the indicator acts to establish the context of the line.

In the *second* entry, an *identifier* (or *name*) is introduced. This can be either a *variable* (such as z, in assumption lines) or a *constant* (say c, in definition lines or primitive lines).

The *third* entry is called the *content*.[2] A content is the symbol '—' when the name is a variable (in assumption lines), and an expression M or the symbol PN in the two other cases.

The *fourth* entry, the *category*, contains an expression A.[3]

The *meaning* of the three kinds of lines can be described shortly as follows: an assumption line contains a *context extension*; a definition line introduces a *definition* and a primitive line introduces a *postulate* (both in a given context).

More explicitly:

(i) *assumption line*: in the empty context (if the indicator is ϵ) or in a context having indicator x, the *assumption* is made that newly introduced variable z has category A; or, in type-theoretic terms, we extend a given context (being empty, or having indicator x) with $z : A$, the *declaration* of variable z of *type* A;

(ii) *definition line*: in the empty context (ϵ) or in a context having indicator x, constant c is *defined* as M of category A;

(iii) *primitive line*: in the empty context (ϵ) or in a context having indicator x, constant c is *postulated* as a primitive notion ('PN') of category A.

In order to clarify these notions, we start with a discussion of contexts in general (Section 2.1) and, in particular, in Automath-style (2.2). This is followed by the description of how formalised definitions (2.3) and postulates (2.4) are dealt with in Automath.

[1]Instead of the semicolon ';' used in Automath, we take the colon ':' – as is common nowadays – for expressing the relation between a content and a category (or 'type').

[2]The content is called the *definition* in the original Automath paper. We use the word *content* to avoid confusion with the general meaning of the word 'definition'.

[3]We shall often use the word 'type' instead of 'category', in order to connect Automath with more common terminology in type theory.

2.1. Contexts

A (type-theoretical) context consists of a list of *declarations* of so-called *declared variables* $x_1, \ldots, x_n$, together with their *types* $A_1, \ldots, A_n$, respectively. In type theory, such a list is generally written as $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$, or, shortly, as $\overline{x} : \overline{A}$.

Before we describe how contexts are treated in Automath (see Section 2.2), we devote some space to two different manners in which contexts are expressed. For that we first give a mathematical example of how contexts are introduced, followed by the same example denoted in flag-style (or Fitch-style; see below).

In mathematics and logic, contexts are omnipresent. See the following example, which illustrates their use:

Example 2.1. (Contexts in mathematics)
LEMMA 5.2 (a) Let A, B and C be subsets of a given set S. Assume
$A \subseteq B$ and $B \subseteq C$. Then also $A \subseteq C$.
(b) Let, moreover, $C \subseteq A$. Then $A = B$.
(c) $A \cap B \subseteq A \cup B$.
(d) Let $A = B$. Then $A \cap B = A \cup B$.

Proof:
Proof of (a): Let x be an element of A. Then $x \in B$ so also $x \in C$. □

The various contexts in this example can be expressed as follows in a typetheoretical style, using the propositions-as-types isomorphism.

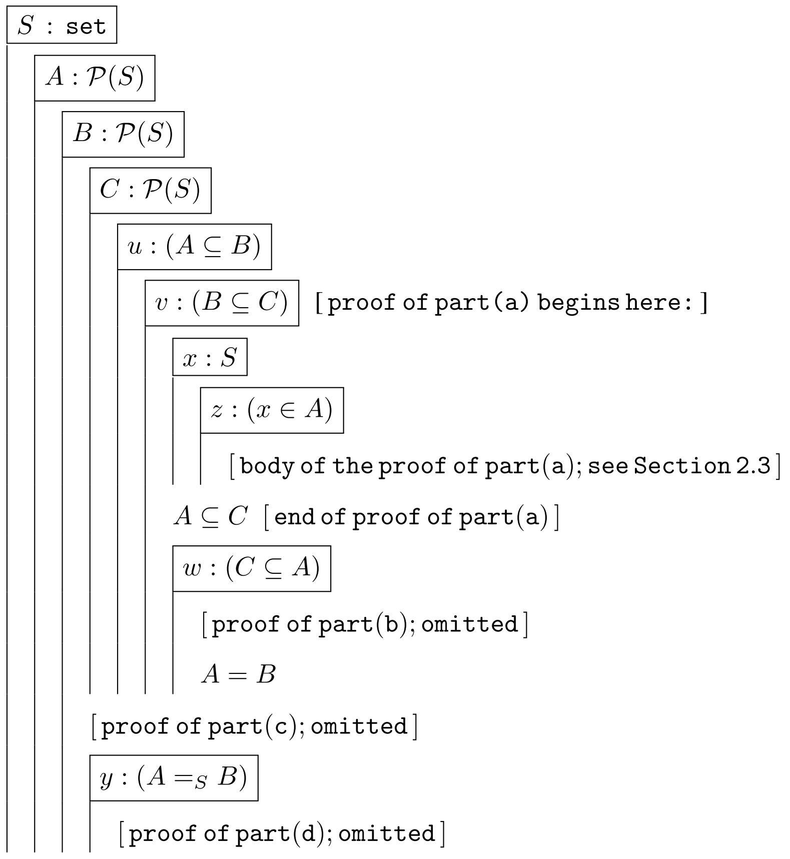
Remark 2.2. We only want to exemplify the notion 'context'. The symbol set represents the 'universe' of sets. We do not explain why we introduce a 'universe' S. We identify sets in the universe by means of the power set of S: a set is an inhabitant of type $\mathcal{P}(S)$. Moreover, we freely employ infix notation, although this is against the Automath conventions. We also deviate a bit from the Automath precision by not mentioning the S when using the symbols $\subseteq$, $=$, $\cap$, $\cup$ and $\in$. For details, see [5].

Context of part (a):
$S : \mathtt{set},\ A : \mathcal{P}(S),\ B : \mathcal{P}(S),\ C : \mathcal{P}(S),\ u : A \subseteq B,\ v : B \subseteq C.$
Context of part (b):
$S : \mathtt{set},\ A : \mathcal{P}(S),\ B : \mathcal{P}(S),\ C : \mathcal{P}(S),\ u : A \subseteq B,\ v : B \subseteq C,\ w : C \subseteq A.$
Context of part (c):
$S : \mathtt{set},\ A : \mathcal{P}(S),\ B : \mathcal{P}(S).$
Context of part (d):
$S : \mathtt{set},\ A : \mathcal{P}(S),\ B : \mathcal{P}(S),\ y : (A = B).$
Context of PROOF of part (a):
$S : \mathtt{set},\ A : \mathcal{P}(S),\ B : \mathcal{P}(S),\ C : \mathcal{P}(S),\ u : A \subseteq B,\ v : B \subseteq C,\ x : S,$
$z : (x \in A).$

It will be clear that the type-theoretical style requires a frequent repetition of initial parts of contexts, in particular for the tree-like presentation as is known from proof theory. Such repetitions are to a great extent avoided in the linear presentation as is common in mathematics texts (see the Lemma above).

In formal versions of mathematics, the duplication of context elements is unpleasant. One way out is to use a *Fitch-style* notation[4] (see [10]). Fitch introduced this notation in his logical course, with the clear intention to condense equal parts of contexts. He uses a linear presentation to derive logical proofs, embedded in nested rectangles to indicate the span of an assumption: each initial entry of a rectangle is supposed to be an assumption – i.e., a context entry – in all (and only) the statements within that rectangle, so also in the deeper nested ones. This gives a *block structure* to texts, as is well-known from programming languages such as Algol. For an overview of how this notation is used in systems of natural deduction, see [11].

A variant of the Fitch-notation is the so-called *flag notation*, introduced in the 1970's, independently from Fitch, by R.P. Nederpelt and N.G. de Bruijn. It uses a *flag* for each initial entry of a block (acting as an assumption), and a *flag pole* to indicate the range of that assumption (and thus the length



S : set
$A : \mathcal{P}(S)$
$B : \mathcal{P}(S)$
$C : \mathcal{P}(S)$
$u : (A \subseteq B)$
$v : (B \subseteq C)$
[proof of part (a) begins here:]
$x : S$
$z : (x \in A)$
[body of the proof of part(a); see Section 2.3]
$A \subseteq C$ [end of proof of part(a)]
$w : (C \subseteq A)$
[proof of part(b); omitted]
$A = B$
[proof of part(c); omitted]
$y : (A =_S B)$
[proof of part(d); omitted]

Figure 3. Example of a derivation in flag style

[4]This presentation of natural deduction already appears in the work of Stanisław Jaśkowski ([9]).

of the block). In Figure 3 we express the structure of the above example lemma in flag format, in order to show how it works. (See also [5].)

It is obvious that the flag- (or Fitch-)notation allows *sharing* of declarations: a comparison with the series of example contexts presented above, shows that one flag can cover several context items, so repetitions have been avoided.

In the example we illustrated that assumptions phrased in a lemma or theorem can be translated as newly introduced context items, that can be withdrawn later. But also the application of *logical* rules may lead to context changes (either *context extension* or *context shortening*). This can be seen, again, in the previous example, where the assumptions $x : S$ and $z : (x \in A)$ are introduced in order to obtain the logical result $A \subseteq C$ (i.e., $\forall_{x:S}(x \in A \Rightarrow x \in C)$)as the final result of part (a). We come back to this in Section 2.3.

2.2. Assumption lines in Automath

Automath goes still further in avoiding repetitions of context assumptions. De Bruijn's idea was to use a handy *naming convention* for contexts: each context is represented by its *last* declared variable, acting as the *indicator* of the complete context. So the four contexts of parts (a) to (d) of the Lemma given in the previous section (cf. Figure 3), are simply referred to as v, w, B and y, respectively.

We shall explain below why this is advantageous. But firstly we note that this naming only works smoothly if we agree never to introduce the name of a declared variable more than once. This *unique naming convention* for context variables is, of course, rather restrictive and cannot be maintained long. However, in theory (and in this paper) we stick to it, noting that de Bruijn found a practical way out in Automath by using a so-called *paragraph system* (see [8], p. 78 or [12], Section 11), thus securing the desired unicity of names for context variables, and for defined constants, as well.

Remark 2.3. (1) In Automath all defined constants are supposed to be different, as well, and taken from another set than variables. This has the same drawbacks regarding unicity, which can be overcome similarly as with context variables.

(2) *Abstraction variables* are named independently of the names of context variables and defined constants. Moreover they do not obey the unique naming convention. For example, the name x in $\lambda x : A.B$ (or in $\Pi x : A.B$) can be multiply used, as long as the ranges of the various x's are not interfering.

In Automath-style, a single assumption $z : N$ can be added as follows to an already known context named x:

$$x \quad * \quad z \quad := \quad \text{—} \quad : \quad N$$

Such a line is called an *assumption line*. In case the context to which $z : N$ is added, is the *empty context* (i.e., if the assumption stands on its own), then the symbol ϵ is used:

$$\epsilon \quad * \quad z \quad := \quad \text{—} \quad : \quad N$$

As an example, we express the contexts belonging to the Lemma parts (a) to (d) mentioned in the previous section, in Automath format.

We start with part (a), in which the context consists of six assumptions:

$$\begin{array}{llllll}
\epsilon & * & S & := & — & : & \texttt{set} \\
S & * & A & := & — & : & \mathcal{P}(S) \\
A & * & B & := & — & : & \mathcal{P}(S) \\
B & * & C & := & — & : & \mathcal{P}(S) \\
C & * & u & := & — & : & A \subseteq B \\
u & * & v & := & — & : & B \subseteq C
\end{array}$$

Now it is easy to express the context for part (b), which apparently consists of only one assumption (the other six are covered by the indicator v):

$$\begin{array}{lllllll}
v & * & w & := & — & : & C \subseteq A
\end{array}$$

The context for part (c) can be described simply by 'plugging in' at the identifier B, which is introduced in the third assumption of part (a). So if we need that context, we just put a B in the 'indicator' column.

This is exemplified in the Automath-version of the context for part (d):

$$\begin{array}{lllllll}
B & * & y & := & — & : & A = B
\end{array}$$

The Automath-notation of contexts is still more 'economic' than the flag notation. Using flags in Fitch-style, a closed block (i.e., the ending of a flag pole) cannot be reopened. Yet, such a thing is regularly the case in mathematics. For example, one could continue Figure 3 with describing another lemma about a triple of sets, A, B and C. But since the flag of C has been closed already, the only way out is to re-introduce a flag with C.

In the Automath-style, this is not needed, even when all flags in Figure 3 have been closed and the text in Figure 1 is far out of sight. One just puts the C as the indicator of a new line and the context named C is available, again. Otherwise said, the flags of S, A, B and C have been reopened.

2.3. Definition lines in Automath

The possibility to add definitions is essential in mathematics. One generally uses a special format for definitions. For example, previous to the above example called 'Lemma 5.2' in Example 2.1 there will have appeared a definition for inclusion of sets, for example:

Definition 2.4. Let A and B be sets. One says that A is included in B (denoted $A \subseteq B$) if for all $x \in A$ it holds that $x \in B$.

Note that this definition needs a *context* again, namely $S : \texttt{set}$, $A : \mathcal{P}(S)$, $B : \mathcal{P}(S)$. This is the context named B in the previous section.

In Automath, we use a line of the form (ii) for stating a definition. Using the context named B, the definition above can be expressed as follows:

$$\begin{array}{lllllll}
B & * & \subseteq & := & \forall x : S \, . \, (x \in A \Rightarrow x \in B) & : & \texttt{prop}
\end{array}$$

Here prop represents the collection of all propositions. Since the symbol '$\subseteq$' is used in the assumption line with identifier u, this definition line should precede it. So it could be placed immediately below the assumption lines for A and B given above.

Remark 2.5. The newly defined constant $\subseteq$ depends on three *parameters*: the context variables S, A and B; in mathematics, using infix-notation, one would rather write $A \subseteq_S B$ in this definition, instead of mere $\subseteq$.

Yet more common is the notation $A \subseteq B$, so without referring to S. It becomes a bit annoying to repeatedly mentioning the 'universe' S in the Automath expressions. A mathematician would hardly ever mention these S's – so leave them implicit. See Section 2.5.2 for how de Bruijn solved this problem.

In mathematics, definitions are mostly separated from, for example, the statement of a theorem or its proof. In Automath, however, definitions are omnipresent. *Every* subject M of a typing statement expressed in an Automath line (i.e., a statement of the form $M : N$, expressing 'M has category N') gets a name, say d. In a context named x, this is written as

$$x \quad * \quad d \quad := \quad M \quad : \quad N$$

(to be read as: 'In the context named x, constant d is defined as M, which has category N').

Note that the earlier given example – the definition of $\subseteq$ – has exactly the same format. We extend this example to a *proof* of the above LEMMA 5.2 (a), as expressed in Example 2.1. In Automath style this reads as below. We hereby assume that the given lines are preceded by the assumption lines forming the context called v, and the definition line for $\subseteq$, given above. We use the derivation rules for the type system λD ([5]). For example, in the definition line introducing d_1 we use that the type of u is $A \subseteq B$, which is definitionally equal to $\forall x : S \,.\, ((x \in A) \Rightarrow (x \in B))$. Then $u\,x\,z$, that is $(u\,x)z$, or u applied to x and the result applied to z, is of type (i.e., 'proves') $x \in B$.

v	$*$	x	$:=$	—	:	S
x	$*$	z	$:=$	—	:	$x \in A$
z	$*$	d_1	$:=$	$u\,x\,z$	:	$x \in B$
z	$*$	d_2	$:=$	$v\,x\,d_1$	:	$x \in C$
x	$*$	d_3	$:=$	$\lambda z : (x \in A) \,.\, d_2$	:	$(x \in A) \Rightarrow (x \in C)$
v	$*$	d_4	$:=$	$\lambda x : S \,.\, d_3$	:	$A \subseteq C$

The lines defining d_1, d_2 and d_3 can be regarded as the 'body of the proof of part(a)' as mentioned in Figure 3.

Note the context structure in this example: the context expands from v via x to z, and then shrinks again via x to v. This can be expressed clearer in the corresponding flag version.

We conclude that in an Automath book, apart from assumption lines and primitive lines (see below), *every* line is a definition line. This is different from what is common in mathematics, but it makes it easy and feasible to write a compact, yet precise formal text. In the above simple example we can already observe this, since we use, e.g., d_1 in the definition of d_2, and so on. Without the definitions of d_1 to d_3, the subject for d_4 should have been written out as

$\lambda x : S \,.\, \lambda z : (x \in A) \,.\, v\, x\, (u\, x\, z)$,

which is considerably longer and further away from a proof as is usual in the 'mathematical style'.

Another observation is that defined constants such as $\subseteq$ (and also d_1 to d_4) in the above case, *when used*, must be provided with a list of *instantiated parameters*. For example, in the definition above, the type of d_4 is $A \subseteq C$. Here A and C are instantiations: in its definition line, $\subseteq$ has the *parameter list* $S,\ A,\ B$. But in the final type, the parameter list is $S,\ A,\ C$. So we have identity substitutions for S and A, but a 'genuine' substitution of C for the final parameter B. This is a simple case; but generally, these substitutions can be quite complicated.

We note that the rules of Automath, being the rules of a type theory, force us to only execute instantiations with expressions of 'the same type' as the original context variables. For example, both B and C have type $\mathcal{P}(S)$. (Often, the instantiation of a context variable requires an analogous substitution of the types; see [5], Section 9.4, for details. For now, we shall not go into this matter.)

2.4. Primitive lines

What remains is to explain what lines of sort (iii) stand for. We call these lines *primitive lines*. They concern the introduction of so-called *primitive notions* (PNs), used in stating axioms or primitive elements of sets. For example, the first two axioms of Peano about the set $\mathbb{N}$ of natural numbers and a number 0 in it, can be phrased as follows in Automath-style:

$\epsilon \quad * \quad \mathbb{N} \quad := \quad \text{PN} \quad : \quad \texttt{set}$

$\epsilon \quad * \quad 0 \quad := \quad \text{PN} \quad : \quad \mathbb{N}$

Here '`set`' stands for the collection of all sets. So now $\mathbb{N}$ is a primitive set and 0 a primitive element of type $\mathbb{N}$. For both notions, the context is empty. Note that this notation preserves the typing relation, so these lines express that $\mathbb{N} : \texttt{set}$ and $0 : \mathbb{N}$. But in these lines 'PN' is written instead of a definiens after the symbol ':=', since both $\mathbb{N}$ and 0 are not defined but axiomatically introduced.

One may also define primitive notions *in a context*. For example, one way of postulating the successor function in Peano-style is as follows (the other way is to introduce s as a function of type $\mathbb{N} \to \mathbb{N}$, in the empty context):

$\epsilon \quad * \quad n \quad := \quad — \quad : \quad \mathbb{N}$

$n \quad * \quad s \quad := \quad \text{PN} \quad : \quad \mathbb{N}$

and one could continue with defining the number 1 (in the empty context):

$\epsilon \quad * \quad 1 \quad := \quad s(0) \quad : \quad \mathbb{N}$

The last three example lines show all three different sorts of Automath lines: an assumption line, a primitive line and a definition line, respectively.

Finally, we note that not only axiomatic *objects*, but also axioms (i.e., axiomatic *propositions*) can be expressed in Automath. For example, one of the Peano-axioms can be expressed as follows:

$n \quad * \quad AX3 \quad := \quad \text{PN} \quad : \quad \neg(s(n) = 0)$

In an average Automath text, assumption lines and definition lines are the most common ones; they 'tell the story'. Primitive lines occur relatively rarely, as one may imagine. For example, the translation (see [8]) of E. Landau's 'Grundlagen der Analysis' ([7]) into Automath contains more than a thousand Automath lines, of which only 26 are primitive ones.

2.5. Special features of Automath

The most important characteristics underlying all Automath languages have now been explained. There are four peculiarities in all these languages that must be clarified. We discuss these below.

2.5.1. λ-abstraction also for type abstraction

In most type systems, the Greek letter λ is used for *functional abstraction* and the capital Π is used for expressing *dependent types*. E.g., $\lambda x : A \,.\, M$ expresses the function sending x of type A to M, and if $M : N$, then the type of $\lambda x : A \,.\, M$ is $\Pi x : A \,.\, N$. In particular, Π can be used for expressing $\forall$ ('universal quantification').

Automath, however, does not differentiate between λ and Π. Both $\lambda x : A \,.\, T$ and $\Pi x : A \,.\, T$ are rendered as $[\,x, A\,]\,T$. So the square brackets $[\ldots]$ have a double role. This is clearly a case of 'overloading', that complicates both the meaning and the syntax of the Automath languages.

As to the meaning: the reader of a text must decide for each $[\ldots]$-abstraction whether it is a usual 'functional' one (a 'λ'), or that it is meant to represent a type abstraction (a 'Π'). This is in most cases not very difficult to decide (cf. [13], Section 6.1), but sometimes it asks for special attention.

On the syntactic level, when Π is identified with λ as in Automath, we have a difficulty when embedding Automath into the Calculus of Constructions. We have the choice to map $[\,x, A\,]\,T$ to either $\lambda x : A \,.\, T$ or $\Pi x : A \,.\, T$, and in CC there are different rules for typing a λ-abstraction or a Π-abstraction. Automath has only one rule for this purpose, but has the extra notion 'type inclusion' (see [14], Section 4.4) to mend the ambiguity. Type inclusion allows e.g. to move from $\lambda x : A \,.\, s_2$ to s_2, and thereby one can simulate one abstraction rule with the other.

2.5.2. Incomplete parameter lists

Consider a line of sort (ii) in Automath, so a definition line (l):

$$x_n \;\; * \;\; c \;\; := \;\; M \;\; : \;\; N \,.$$

Then both M and N may depend on the variables in the list $x_1, \ldots, x_n$ referred to by the indicator x_n. So, the defined constant c also (but implicitly) depends on its *parameter list* $x_1, \ldots, x_n$.

Each case this c, *defined* in line (l), is *used* in a following line (m), one must *instantiate* (i.e., *explicitly* write out) the expressions $L_1, \ldots, L_n$ that have to be substituted for the free variables $x_1 \ldots x_n$. As we saw before, the result is written as $c(L_1, \ldots, L_n)$. The list $L_1, \ldots, L_n$ is called the *instantiated parameter list* corresponding to that instance of c.

Now it frequently occurs that such a line (m) 'below' line (l) has a context that starts with an initial part of the context of c, say $x_1 : A_1, \ldots, x_i : A_i$. Then it regularly happens that $L_1 \equiv x_1, \ldots L_i \equiv x_i$, so the first i substitutions for the context variables are *identity substitutions*. This often occurs not only in one Automath line, but in a full bundle of lines, and also in other circumstances. In that case it becomes a real burden to mention those initial $x_1, \ldots, x_i$ ever and ever again.

Therefore, Automath has the following *parameter omitting* convention: an *initial* part of a parameter list may be omitted when it consists of identity substitutions only. So in the above case, the expression $c(x_1, \dots, x_i, L_{i+1}, \dots, L_n)$ may be written as $c(L_{i+1}, \dots, L_n)$.

Obviously, the latter parameter list of c is *too short* to be syntactically correct. But this is easy to repair: if i items are missing in the parameter list, then the first i declared variables from the context should be added in front (in the same order) to make it correct.

This convention is very convenient when developing mathematical theories in Automath, since it prevents lots of annoying repetitions of context declarations. So 'parameter omitting' is very practical.

In Automath, this convention also applies to *primitive constants*, having been defined in a primitive line.

2.5.3. β-reduction and η-reduction

As usual in many type systems, also in Automath *β-reduction* and *β-conversion* are used to cover the notions of function application and convertibility. And again as usual, two terms in Automath that are acceptable and β-convert to each other, or are 'definitionally equal', may be substituted for each other, without explicit justification. (In our example proof in Section 2.3 we showed this, when silently replacing the type of u, given as $A \subseteq B$, by its definiens $\forall x : S \,.\, ((x \in A) \Rightarrow (x \in B))$, in order to obtain uxz as inhabitant of $x \in B$.)

We recall that β-reduction is the compatible closure of the relation expressing 'applying a function to an argument', i.e., $(\lambda x : M)N \to_\beta M[x := N]$, where $M[x := N]$ is the result of substituting N for all free x's in M. Conversion is the equivalence relation generated by β-reduction. Since λs and Πs are identified in Automath, β-reduction and β-conversion also apply to Π-types. Such Π-types may be provided with arguments in Automath, which is very uncommon in other type theories, but a consequence of the λ-Π-identification.

Automath not only has β-reduction, but also allows *η-reduction* of terms. This corresponds to the 'extensionality' relation on λ-terms, generated by the reduction $(\lambda x : A \,.\, Nx) \to_\eta N$ (if x is not a free variable in N). We note that in Jutting's Automath-version of Landau's book [7], only two η-reductions are necessary, whereas over 6.000 β-reduction take place in the checking procedure and more than 20.000 definition unfoldings ('*δ-reductions*'). Again, see [8].

2.5.4. Argument precedes function

De Bruijn advocates inversion of the usual order function–argument in a formal treatment of mathematics. Instead of fa for 'f applied to a' (the usual notation in λ-calculus), he prefers $< a > f$ and uses this in the Automath languages. He motivates this as follows: an abstraction is written in front (M becomes $(\lambda x : A) \,.\, M$ when abstracted over A), so it is consistent to do the same with an application (M becomes $< A > M$ when applied to A).

This notation for application has both technical and philosophical advantages, which we shall not mention here. See e.g. [15]. But it is unusual and may be confusing to work with.

2.6. Interpretation of the example text

Now it is easy to 'read' Jutting's Automath text as reproduced in Figure 1. Below we give its translation in common mathematical language. As in Automath, we use round brackets for instantiation and pointed brackets for application. Some explicit notes are added between square brackets. Note that the translation doesn't give many clues about the variations in the contexts of the various lines, whereas the Automath text is precise about these matters.

(1) Let A be a proposition

(2) and let also B be a proposition.

(3) Then *IMP* is defined as the function space $\lambda x : A \,.\, B$. [This λ must be read as Π, so *IMP* (in the context of A and B) can be considered to embody the implication $A \Rightarrow B$.]

(4) Let moreover C be a proposition,

(5) assume that *IMP*(A, B) holds with proof object I

(6) and assume that *IMP*(B, C) holds with proof object J.

(7) Then $\lambda X : A \;.\; \langle\langle J\, I\rangle X\rangle$ proves *IMP*(A, C) and is called *TRIMP*. [λ represents Π, again. See also Section 2.5, (4).]

(8) By axiom, *CON* is a proposition, in the empty context. [It formalizes the notion of 'contradiction'.]

(9) In the context of A, constant *NOT* is defined as *IMP*(*CON*). [To be read as *IMP*$(A,$ *CON*$)$, since the given parameter list is incomplete.]

(10) In the same context, *WEL* is defined as *NOT*(*NOT*(A)).

(11) Assume that proposition A is inhabited by $A1$,

(12) then $\lambda x :$ *NOT*$(A) \,.\, \langle X\; A1\rangle$, called *WELI*, proves *WEL*(A) [so lines (11)and (12) together express: if A holds, then also $\neg\neg A$].

(13) Assume that *WEL*(A) is inhabited by W,

(14) then, by axiom, A is inhabited by *ET* ['excluded third'].

(15) Assume that *CON* is inhabited by $C1$,

(16) then *ET*$(\lambda x :$ *NOT*$(A) \,.\, C1)$ proves A. [Note: the parameter list of *ET* is incomplete.]

3. Automath and modern type theory

In this section we compare the essence of the syntax of Automath (which we called the 'generic' syntax) to that of another type system: the *Calculus of Constructions* of Coquand and Huet (*CC*, also called λC). Cf. [1]. This CC is a very influential type theory, both for theoretical purposes as in practice. It is, for example, the basis of the theorem prover Coq (see [16]). CC has been devised around 1984, many years later than Automath (dating from 1968). However, Coq is still in development (see [2]), but the Automath work has come to an end. The standard reference on type systems is [17].

Coquand and Huet pay tribute to de Bruijn's Automath [18], but the original syntactic rules of Automath are rather complicated and difficult to compare with CC's elegant syntactic rules. An early example in this direction can be found in [14], 5.5.4. An extension of CC with definitions, such as the type system λD described in [5], has more direct links to Automath. But the direct relation of λD to Automath that we show below, has not yet been described.

3.1. Abandoning inessential Automath features

First we recall that the purpose of N.G. de Bruijn when developing Automath, was to produce a *practical* system, immediately useable for a mathematician to formalize and verify mathematical content. A magnificent example to show that de Bruijn attains his goals, is the impressive translation of the complete book *Grundlagen der Analysis* ('Foundations of Analysis') of E. Landau [7], into Automath, by L.S. van Benthem Jutting. The translation of the book is well readable after a little exercise, although it covers 13.433 Automath lines. (In the previous section we gave a very short impression of this translation.)

For the conversion of the Automath syntax to a CC-like version, human-readability is less important then syntactical simplicity. That's why we *abandon* some complicating features of Automath (cf. Section 2.5) that are a bit unusual, or only meant for the human user, namely:

(1) *λ-abstraction also for type abstraction* This Automath-peculiarity is not very attractive and unnecessary ([13]). We restore Π-abstraction in our CC-like derivation rules.

(2) *Incomplete parameter lists* These are desirable for humans, but needless for theoretical analysis. So we abandon them.

(3) *η-reduction* This can be harmlessly omitted, as suggested already in Section 2.5.3.

(4) *Argument precedes function* This is an Automath-peculiarity that is debatable because it is so uncommon. We restore the usual order. We also employ the usual λ-calculus notation. So we write *application* of F to M as $F\,M$ (with an invisible operation sign between function and argument), instead of $< M > F$ in Automath. Note that we also use the notation with round brackets, $G(N)$, but only for *instantiation*. So $G(N)$ has the meaning: 'constant G with N substituted for the context variable'.

3.2. Automath judgements

We have discussed the three kinds of Automath lines: *assumption lines, definition lines and primitive lines*. Now we want to give a type system for establishing *correctness*. Note that, in an Automath text, the lines are not given as a set, but as an ordered *list*. Such a list is called a *book* in the original Automath report. An Automath book is *correct* if it can be derived by means of the rules of Automath.

There are different correctness requirements for the three sorts of lines in Automath. First note that the lines in a correct Automath book depend on earlier ones. We mention two examples: (1) an existing context can be extended or reopened later by referring to its context indicator; (2) a defined constant can be used in every line following its definition, providing that it has been properly provided with instantiations for its parameters.

These dependencies are enabled by the ordered character of each Automath book: every line may depend on any number of *previous* lines. If lines (1) to (l) form a correct (and ordered) book $\mathbf{B}$, then we may add a new line $(l + 1)$, provided that it is correct with respect to $\mathbf{B}$. So books are developed *incrementally*, line by line. (This linear build-up is not in contradiction with the tree-like build-up in common type systems, as we sufficiently explained in [5].)

We express correctness not as in the original Automath report ([3]), but by means of three specific rules. We refer to the system λD, as described in [5], to obtain full correctness of an Automath text.

The system λD is a natural extension with definitions of λC (which is an alternative name for CC). We recall the following important notion used in type systems as λC (cf. [17] or [19]).

A type-theoretical *judgement* has the form

$\Gamma \vdash M : N$,

expressing that it is derivable ('$\vdash$') in context Γ that M has type N. Here, a context is a list of variable declarations $x_1 : A_1, \ldots, x_n : A_n$.

Since we have to do with an Automath book, we also have to consider a context Δ of definitions. So in order to establish correctness, we need for Automath some sort of *extended judgement*. This has in λD the following form:

$\Delta\,;\,\Gamma \vdash M : N$,

meaning that, with respect to definition context Δ and context Γ, we can derive that M has type N. Here, a definition context is a list of definitions, each of the form $c(x_1 : A_1, \ldots, x_n : A_n) := M : A$. (A lemma about the deduction rules implies that Δ and Γ have already been justified in that case; see again [5].)

3.3. Coherent Automath books

We start with a formal description of an Automath *book*, referring back to notions introduced in Section 2.

The following notions are borrowed from type theory (see e.g. [17]): *Var* is an infinite set of *variables*; *Const* is a disjoint, also infinite set of *constants*; *Term* is the set of *pseudo-terms* of λD, given as usual by an inductive definition.

Term includes *Var*, *Const*(*Term*, ..., *Term*), $\lambda\,Var : Term\,.\ Term$ (the so-called λ-terms), $\Pi\,Var : Term\,.\ Term$ (the Π-terms) and (*Term Term*) (the application terms).

Definition 3.1. (Line; book)

1. An *Automath line* is either an *assumption line*, a *definition line* or a *primitive line*. See Figure 2 for the appearance of each of these lines and for the various notions connected with Automath lines, e.g.:
   An *indicator* is ϵ or an $x \in Var$, an *identifier* is an $x \in Var$ or a $c \in Const$, a *content* is either '$-$' or 'PN' or an $M \in Term$, and a *category* (or *type*) is an $A \in Term$.
2. A sequence of Automath lines is called a *book*, denoted $\mathbf{B}$.

We give some terminology around lines and books:

Definition 3.2. ($\in$, $+$, $<$, $\subseteq$; **precede, subbook)**

1. $l \in \mathbf{B}$ means that line l occurs in book $\mathbf{B}$.
2. Concatenation of lines and/or books is denoted by the symbol $+$, e.g., $\mathbf{B_1} + \mathbf{B_2}$ or $\mathbf{B_1} + l$.
3. We say that line l *precedes* line m in book $\mathbf{B}$, denoted $l < m$, if $\mathbf{B} \equiv \mathbf{B_1} + l + \mathbf{B_2} + m + \mathbf{B_3}$.

4. We say $\mathbf{B}'$ is a *subbook* of $\mathbf{B}$, and write $\mathbf{B}' \subseteq \mathbf{B}$, if all lines in $\mathbf{B}'$ also appear in $\mathbf{B}$ and inherit the precedence order.

A first step to correctness concerns the structure of Automath texts as regards the use of *identifiers* introduced in an Automath book. First, we prevent ambiguities in referencing, which occur when an identifier is introduced multiply. Therefore we require that each identifier is *unique*. (This restriction can be eliminated, e.g. by a paragraph system; cf. Section 2.2.)

We also must ensure that no indicator occurring in an Automath line does 'dangle': we recall that each context indicator z must refer to precisely one 'earlier' introduced identifier of the same name, in order to establish exactly which context is 'named' by z. This earlier identifier should pop up in an assumption line.

An Automath book in which the last-mentioned conditions hold, we call a *coherent* book.

Definition 3.3. (Coherent book; $\prec$))

1. The book $\mathbf{B}$ is called *coherent* if the following hold.
   (a) All identifiers of lines in $\mathbf{B}$ are different.
   (b) Each context indicator y in a line $m \in \mathbf{B}$, with $y \not\equiv \epsilon$, appears as *identifier* in an *assumption* line l of $\mathbf{B}$ where $l < m$.
2. Let m be a line in a book $\mathbf{B}$, with indicator y and identifier z. If $\mathbf{B}$ is coherent and $y \not\equiv \epsilon$, then we say that $y \prec z$.

We have the following easy properties. If m is a line in a coherent book $\mathbf{B}$ with identifier z and indicator $y \not\equiv \epsilon$, then there is exactly one line $l < m$ in $\mathbf{B}$ with identifier y. So, for each identifier z in a coherent book $\mathbf{B}$, we can 'follow the trail back' to collect the full chain of variables that 'precede' z according to the relation $\prec$. Note that such a chain is always finite and consists of variables, but for the 'greatest' element, which may be a constant.

If such a z *is* a variable (not a constant), we recursively define the notion γ_z as the full list of the variables in this chain, including z itself. We simultaneously define the typing-context Γ_z of z, being γ_z provided with typing information for each of the variables in it.

Definition 3.4. (Subject-list γ_z; typing-context Γ_z)
Let $\mathbf{B}$ be a coherent book and let *assumption* line m be in $\mathbf{B}$. We proceed by induction on the relation '$\prec$', taking it that γ_ϵ is the empty list and Γ_ϵ the empty context. Let $m \equiv y * z := \text{—} : A$, where y may be ϵ. If $y \not\equiv \epsilon$, then $y \prec z$, so γ_y and Γ_y are already known by induction. Now we define:

(1) The *subject-list* γ_z of z is $\gamma_y,\ z$.

(2) The *typing-context* Γ_z of z is $\Gamma_y, z : A$.

3.4. The derivation rules for Automath

We will now define what a *well-formed book* is in the Automath sense, using the system λD to obtain the desired type-correctness for the terms. As λD contains rules for definitions, it fits very well with Automath, but basically any other type theory with formal definitions could be used. This is because the book structure with lines, built up in the way described in the previous section, is largely orthogonal to the notion of *well-typedness*, that we need to judge whether a new line can be added to a book.

We define by induction on $\mathbf{B}$ the notion $\mathbf{B}\ ok$, expressing that $\mathbf{B}$ is well-formed. The definition splits into three parts, depending on whether we add *(i)* an assumption line, *(ii)* a definition line or *(iii)* a primitive line to a book $\mathbf{B}$, which we assume to be already *ok*. The new book that we obtain after the addition of one of the three lines, we call $\mathbf{B}^+$. Simultaneously we define $\Delta_{\mathbf{B}}$, the translation of the well-formed book $\mathbf{B}$ into a λD-'book' (see [5]).

We recall that the symbol s stands for a so-called *sort*, being either $*$ or $\Box$. These things have been described and explained in [17]. See also, again, [5].

Definition 3.5. (*ok*; translation $\Delta_{\mathbf{B}}$)

1. *Start of the induction:*
   - the empty book ϵ is ok
   - $\Delta_\epsilon := < >$
2. (i) If $\mathbf{B}^+ = \mathbf{B} + (z * x := \text{—} : A)$, i.e., *adding an assumption line to* $\mathbf{B}$, then
   - $(\textit{assum})$ $\dfrac{\mathbf{B}\ ok \qquad \Delta_{\mathbf{B}}\,;\Gamma_z \vdash A : s}{\mathbf{B}^+\ ok}$ if x is $\mathbf{B}$-fresh
   - $\Delta_{\mathbf{B}^+} := \Delta_{\mathbf{B}}$

   (ii) If $\mathbf{B}^+ = \mathbf{B} + (z * c := M : A)$, i.e., *adding a definition line to* $\mathbf{B}$, then
   - $(\textit{defin})$ $\dfrac{\mathbf{B}\ ok \qquad \Delta_{\mathbf{B}}\,;\Gamma_z \vdash M : A}{\mathbf{B}^+\ ok}$ if c is $\mathbf{B}$-fresh
   - $\Delta_{\mathbf{B}^+} := \Delta_{\mathbf{B}}\,, (\Gamma_z \rhd c(\gamma_z) := M : A)$

   (iii) If $\mathbf{B}^+ = \mathbf{B} + (z * c := \text{PN} : A)$, i.e., *adding a primitive line to* $\mathbf{B}$, then
   - $(\textit{prim})$ $\dfrac{\mathbf{B}\ ok \qquad \Delta_{\mathbf{B}}\,;\Gamma_z \vdash A : s}{\mathbf{B}^+\ ok}$ if c is $\mathbf{B}$-fresh
   - $\Delta_{\mathbf{B}^+} := \Delta_{\mathbf{B}}\,, (\Gamma_z \rhd c(\gamma_z) := \text{PN} : A)$

The use of λD as an 'auxiliary type system' to guarantee that the lines added to the book contain well-formed (and well-typed) expressions, conforms with the philosophy of de Bruijn: he viewed the type checking of the expressions that were added to a book as a purely algorithmic issue. In his 1970 paper, de Bruijn introduces the book-concept and describes in an algorithmic way what should be verified (i.e., type-checked) before a line can be added. Our Definition 3.5 mimics this via λD-derivation rules.

3.5. Automath and type theory

In type theory, assumptions (or 'declarations', in type-theoretic words) are usually collected into 'contexts' $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$, preceeding a 'judgement'. But in an Automath book $\mathbf{B}$, every new assumption is written in a separate line. Therefore, it may easily happen that an assumption turns out to be *redundant*, to be explained below. We firstly define:

Definition 3.6. (dead-end assumption line)
An assumption line $l \;\equiv\; z * x \;:=\; \text{—} \;:\; A$ in a book $\mathbf{B}$ is a *dead-end* when identifier x is not an indicator of any line following l in $\mathbf{B}$.

One might say that a dead-end assumption line has no 'successors' in the book. Of course, there may be several dead-end lines in a book $\mathbf{B}$. It will be clear that each dead-end line can be removed from the book without influencing any of the other lines. But note that, after removal of a dead-end-line, another dead-end line can turn up that first was 'hidden.

Example 3.7. Assume that lines l_1 and l_2 appear in $\mathbf{B}$, with $l_1 < l_2$ and such that

$$l_1 \;\equiv\; z * y := \text{—} : A \quad \text{and} \quad l_2 \;\equiv\; y * x := \text{—} : B.$$

Now, let l_2 be a dead-end assumption line in $\mathbf{B}$ and let's remove this l_2. Then line l_1 may become a new dead-end in $\mathbf{B}$ (which is was not before). This happens if identifier y does not occur as indicator in another line than the now removed l_2.

It is not hard to show that one may successively remove *all* dead-ends from a given book $\mathbf{B}$, also the newly popped-up as described just now. All these lines together are called the *redundant* assumption lines of $\mathbf{B}$. It also holds that there is a *unique* final result, independent of the *order* of the removals of dead-ends. (Of course, 'new' dead-ends such as the above l_2 can only be removed neatly *after* the removal of the 'old' ones such as l_1.) Moreover, in each step of this process of removals, one coherent book is replaced by another coherent one, and an *ok*-book by a new *ok*-book.

Given a coherent book $\mathbf{B}$, we call the result of eliminating all dead-ends from it a *clean book*.

Definition 3.8. (clean book; $\texttt{cl}(\mathbf{B})$)

1. A coherent Automath book $\mathbf{B}$ in which no assumption line is a dead-end, is called a *clean* book.

2. The clean book being the result of eliminating all redundant lines from a book $\mathbf{B}$, is denoted $\texttt{cl}(\mathbf{B})$.

The following theorem links an Automath book $\mathbf{B}$ to the type-theoretic book $\Delta_{\mathbf{B}}$. It follows directly from the rules and the definitions in Definition 3.5. The correspondence between the two kinds of books is obvious and doesn't need further discussion.

Theorem 3.9. Let $\mathbf{B}$ be an *ok* Automath book.

1. Then $\Delta_{\mathbf{B}}$ is a 'legal environment' in λD, in the sense as defined in [5], Def. 10.4.

2. Moreover, the definition lines of $\mathbf{B}$ correspond one-to-one with the judgements in $\Delta_{\mathbf{B}}$, in the sense that the sequence of constants in the definition lines of $\mathbf{B}$ is the same as the sequence of constants in the judgements occurring in $\Delta_{\mathbf{B}}$. Also, the sequence of typing expressions $M : A$ in the definition lines $\mathbf{B}$ is the same as in the judgements of $\Delta_{\mathbf{B}}$, and a similar thing holds for the typing property PN : A.

3. If $\mathbf{B}$ is a *clean* book, then all assumption lines with identifier x of $\mathbf{B}$ occur in the contexts of exactly those judgements for which the constant c depends on x (i.e., $x \in \gamma_y$, where y is the indicator of the c-line in $\mathbf{B}$.

So there is a strong connection between a clean Automath-book $\mathbf{B}$ and the corresponding type-theoretic book $\Delta_{\mathbf{B}}$. It is not hard to see that all relevant 'information' in $\mathbf{B}$ (i.e., the information in $\mathtt{cl}(\mathbf{B})$) has a counterpart in $\Delta_{\mathbf{B}}$. Note that the image $\Delta_{\mathbf{B}}$, being an environment in λD, consists of definitions only.

In the transition from $\mathtt{cl}(\mathbf{B})$ to $\Delta_{\mathbf{B}}$, the nice 'sharing' of context declarations $x : A$, an asset of Automath, is completely undone. Every definition in $\Delta_{\mathbf{B}}$ has its own context Γ, leading to numerous repetitions of declarations and no sharing at all.

The lack of sharing is also obvious in the simplest translation the other way around, leading from a legal environment Δ to an Automath book $\mathbf{B}_{\Delta}$. In this translation, the image of a definition in Δ, e.g., $\overline{x} : \overline{A} \rhd a(\overline{x}) := M/\text{PN} : N$, becomes a list of n assumption lines ending in one definition line or primitive line (here n is the number of declarations in $\overline{x} : \overline{A}$):

$$\begin{array}{llllll}
\epsilon & * & x_1 & := & — & : & A_1 \\
x_1 & * & x_2 & := & — & : & A_2 \\
\vdots & & & & & & \\
x_{n-1} & * & x_n & := & — & : & A_n \\
x_n & * & a & := & M/\text{PN} & : & N
\end{array}$$

In the desired translation from Δ to $\mathbf{B}_{\Delta}$, one replaces the lines in Δ, one by one and in the same order, by clusters of lines as given just now. Of course, in this process one probably has to rename variables and constants in order to keep all variables and constants distinct.

Lemma 3.10. The translation described above maps a legal λD-environment Δ onto an *ok* and clean Automath book $\mathbf{B}_{\Delta}$.

(1) $\quad * \quad A \quad := \quad — \quad ; \quad$ *Prop* $\longrightarrow$ *Check:* $\emptyset ; \emptyset \vdash$ *Prop* : *Type*
(2) $\quad A \quad * \quad B \quad := \quad — \quad ; \quad$ *Prop* $\longrightarrow$ *Check:* $\emptyset ; A :$ *Prop* $\vdash$ *Prop* : *Type*
(3) $\quad B \quad * \quad$ *IMP* $\quad := \quad \Pi x : A \,.\, B \quad ; \quad$ *Prop* $\longrightarrow$ *Check:* $\emptyset ; A, B :$ *Prop* $\vdash \Pi x : A \,.\, B :$ *Prop*
(4) $\quad B \quad * \quad C \quad := \quad — \quad ; \quad$ *Prop* $\longrightarrow$ *Check:* *IMP*$(A, B) := \Pi x : A \,.\, B :$ *Prop* ; $A, B :$ *Prop* $\vdash$ *Prop* : *Type*
(5) $\quad C \quad * \quad I \quad := \quad — \quad ; \quad$ *IMP*(A, B) $\longrightarrow$ *Check:* *IMP*$(A, B) := \Pi x : A \,.\, B :$ *Prop* ; $A, B, C :$ *Prop* $\vdash$ *IMP*(A, B) : *Prop*

Figure 4. Automath text with checking obligations

It is not hard to find a procedure to recover the sharing of variables after the translation. We don't elaborate on this.

3.6. Example of the verification of an Automath book via a typed lambda calculus

We show how the first five lines of the Automath text of Figure 1 from van Benthem Jutting's formalisation of Landau's book, is a 'correct' book in the sense of Definition 3.5. See Figure 4.

In this example, we write the text line by line and indicate in between the λD-typing judgement that is checked. We omit the '*ok*'-word, but just incrementally add lines. We write, as nowadays usual, $\Pi x : A \,.\, B$ instead of $[x : A]B$. Note for example, that line (5) may be added to the book consisting of lines (1)—(4), because the judgement following line (5) in Figure 4 – the one to be checked – can be derived in type theory (e.g., λD). (This judgement does not belong to the Automath text, it is added just for the example.) For the exact derivation rules concerning the derivation in λD, see again [5].

4. Conclusion

4.1. De Bruijn's philosophy concerning Automath

As we have discussed already in the first part of this paper, N.G. de Bruijn only allows three species of lines in an Automath book. In our terminology in Section 3, these are recorded in the conclusions of the rules *(assum)*, *(defin)* and *(prim)*: each of these statements enables the extension of the *ok*-book $\mathbf{B}$ with one of the three kinds of Automath lines to generate a new *ok*-book, $\mathbf{B}^+$.

One might wonder where the justifications of the judgements in the *premisses* of these rules are registered, the λD-judgements in Definition 3.5. These judgements are not part of an Automath book. To explain this, we should understand de Bruijns original ideas, since he had a number of strong opinions about the formalization of mathematics and the desirability of such a formalization.

(1) Right from the actual start of the computer age, in the 1960's, N.G. de Bruijn recognised *the potential of computer use for different kinds of mathematical work* – and in particular for verification of mathematical content. That ignited his drive to develop a formal language for mathematics, since he realised that computer aid could considerably enhance the credibility of verification. His early attempts in this direction resulted in the development of a formal language for mathematics, which he called Automath ('mathematics aided by an automaton').

(2) Being a genuine (and very respected) mathematician, it is needless to say that de Bruijn produced a meticulous description of the syntax of Automath. However, *he did not shun computer assistance* in the actual verification of Automath syntax.

For that purpose, he asked computer scientist I. Zandleven to write a large computer programme, with which indeed Jutting's voluminous Automath translation of Landau's analysis textbook ([7]) has been checked. In essence, this Automath programme verified the derivability of the checking requirements. Indeed, de Bruijn's opinion was that such *'administrative' work could well be left to a computer*. And the more so, since a bit later it was proved that the Automath syntax could be seen as a *decidable* derivation system.

(3) In de Bruijn's philosophy, an Automath book should be formal and formally verifiable, but *it should follow the original mathematical reasoning as closely as possible*. At the same time, Automath should be *as simple as possible*. Therefore he endeavoured to try to limit himself, and he attempted to do the job with assumptions and definitions only. He realised, of course, that mathematics offers more than these two 'linguistic categories'. For example, the majority of mathematical texts consists of *statements*, in order to bring the reasoning a step further. De Bruijn's invention was to *extend* all usual mathematical statements to a definition (cf. what we said about this in Section 2.3) and to also treat axioms or axiomatic notions as a kind of definitions ('primitive notions').

(4) Thus he could achieve his goal to *limit himself to only assumption lines, definition lines and primitive lines* in the formal Automath texts, and yet stay close to the mathematical written discourse in its usual layout and order. On a small scale, this is easily visible in our mathematical-style phrasing of the tiny part of Jutting's original Automath text, pictured in Figure 1, when comparing this to the 'mathematical' text in Section 2.6. Looking back to the central part of the present paper, we might say that a *human* should compose the Automath text and that the *computer programme* mainly works on checking the *type-theoretic premisses* of the three main derivation lines (see Section 3.4), inspecting the Automath lines one by one.

(5) A remark in this respect is, that despite the decidability, the computer program used in the seventies sometimes had considerable problems with its verification task, in particular because of the (occasionally almost forbidding) size of the 'search space' involved in the definitional equality checking procedure. In such cases, de Bruijn had no problem with an intervention, amounting to an auxiliary lemma (plus proof!) or an extra definition, in order to *bridge the verifiability gap between one Automath line and another one*. All this fitted smoothly in the Automath syntax, so he saw no objection in that little help. Such a small intervention was incidentally executed, for example, in the Landau-translation by L.S. Jutting.

4.2. Other proof checkers

At the time when de Bruijn developed Automath, the idea of computer assisted formal proof checking arose at various other places. There has been some contact between these developments, but mostly these were done independently. Only later, in approximately the 1980s, there was more international exchange on the ideas and concepts underlying these projects, and the developers sometimes used ideas from each other or developed new systems by combining features from various existing ones.

We will not provide an overview of proof assistants here, but give a brief overview of related projects and how they compare with the 'book' approach that de Bruijn took with Automath. For a historical overview of proof assistants and a more in depth discussion of the ideas, see [20]; for a concrete picture of what present day proof assistants are and how one interacts with them, see [21], where 17 provers are compared by showing the formalization of the basic mathematical result that $\sqrt{2}$ is irrational.

4.2.1. Mizar

The Mizar system (see [22]) has been developed by Andrzej Trybulec and his co-workers since around 1970, with as aim to create a library of formalised mathematics. The user enters a mathematical text (definitions, statements, proofs) and the system checks that text by verifying the logical steps made in the proof and by checking the definitions for syntactic correctness.

The idea is close to de Bruijn's idea of a *mathematical vernacular*, where the mathematics that is entered into the system should be close to what one would write in a book or an article. The idea that the checking is going on 'behind the scene', while the user feeds new mathematics line-by-line, is similar to the book approach of Automath, but in Mizar this is usually done in a 'batch' process: a user writes an article and feeds that as a whole to Mizar, and then it is the system that reports which logical steps it can't follow. Mizar has existed continuously since the 1970s and has developed a very large library of formalised mathematics, *MML*.

4.2.2. HOL

The HOL proof assistants (see [23]) are based on the LCF approach ([24]) due to R. Milner in the beginning of the 1970s. The idea is to have an abstract data type of theorems, where the only closed terms of this data type correspond to provable formulas in higher order logic. (Basically, this logic is the higher order predicate logic as defined by A. Church, ([25]), based on his simple theory of types.) The functions one can write for producing terms of type 'theorem' correspond to the derivation rules for high order logic, which guarantees the validity.

The approach is quite different from de Bruijn's, as one can write any function (in the programming language that the type 'theorem' resides in, being ML in the original approach of Milner) to produce terms of type 'theorem'. These are basically proof-tactics, so one does not write proofs in mathematical style, but in programming style. The most used systems in the 'HOL family' are HOL4, HOL-light and ProofPower.

4.2.3. Isabelle

The Isabelle proof assistant ([26]), developed in Cambridge (Paulson) and Munich (Nipkow, Wenzel), is also based on the LCF approach and it caters for various logics to be definable in it. The main used variant is Isabelle/HOL, which is the higher order logic mentioned in 4.2.2. A particular feature of Isabelle is the so called 'Isar mode', which is an input mode for Isabelle similar to Mizar, where one enters mathematical texts in vernacular style, which is then processed by the system: definitions are checked for correctness and stored, and the proof steps are verified. This is again similar to the 'book' approach, with various advanced features giving additional proof automation support.

4.2.4. Coq

In the beginning of Section 3, we already mentioned the Coq proof assistant ([16]; [2]), which is also based on dependent type theory and the propositions-as-types principle, like Automath. Also in the Coq system, the type checking algorithm is behind the scenes, while the user writes mathematics. The definitions are entered and type checked, and stored as definitions, much like in λD, but the proof-terms are created interactively, via tactics, where the typing algorithm assists in filling the gaps in a proof. These are basically 'holes' in a proof term, but, different from Automath, a user never actually observes the proof-term: it is created 'under the hood'. In Coq, one can write functional programs, prove them correct and execute them in the system. A slightly modified version of the type theory of Coq has been implemented in the Lean theorem prover ([27]), with the specific aim of generating interest from the mathematics community to formalize mathematical proofs.

4.2.5. Agda

The typed programming language Agda ([28]) is based on dependent type theory and its underlying type theory is Martin-Löf Type Theory ([29]), which is close to Coq's type theory. The focus is dependently typed programming, but it is also a proof assistant, where one can formalize arbitrary notions from mathematics and prove programs correct, just like Coq. The interaction is quite different from Coq, because one edits proof-terms directly. These are terms-with-holes, where the holes have types that help the user to fill in the holes. The fact that one writes proof-terms directly is closer to Automath, but, different from Automath, the system gives a lot of interactive support for creating these terms (i.e. filling the holes).

References

[1] Coquand T, Huet G. The Calculus of Constructions. *Information and Computation*, 1988. **76**:95–120. doi:10.1016/0890-5401(88)90005-3.

[2] Coq Development Team. The Coq Proof Assistant, Reference Manual. `coq.inria.fr/`.

[3] Bruijn, de NG. The mathematical language AUTOMATH, its usage and some of its extensions. In: Symposium on Automatic Demonstration, Versailles, 1968, volume 125 of *Lecture Notes in Mathematics*, pp. 29–61. Springer, 1970. Reprinted in [4], pp. 73–100.

[4] Nederpelt RP, Geuvers JH, Vrijer, de RC. Selected Papers on Automath. Studies in logic and the foundations of mathematics. North-Holland, Elsevier, Netherlands, 1994. ISBN:0-444-89822-0.

[5] Nederpelt RP, Geuvers JH. Type Theory and Formal Proof: An Introduction. Cambridge University Press, 2014. ISBN:9781107036505. URL `https://books.google.nl/books?id=wzTJBAAAQBAJ`.

[6] Bruijn, de NG. A survey of the project Automath. In: Seldin JP, Hindley JR (eds.), To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism, pp. 579–606. Academic Press Inc., United States. ISBN:0-12-349050-2, 1980.

[7] Landau E. Grundlagen der Analysis. Leipzig (first ed.), Chelsea Publ. Comp. New York (1960, third ed.), 1930.

[8] Benthem Jutting, van L. Checking Landau's "Grundlagen" in the Automath system : Appendices 3 and 4 (The PN-lines; Excerpt for "Satz 27"). In: Nederpelt RP, Geuvers JH, Vrijer, de RC (eds.), Selected Papers on Automath, Studies in Logic, pp. 763–780. North-Holland Publishing Company, Netherlands. ISBN:0-444-89822-0, 1994 (originally 1976).

[9] Jaśkowski S. On the Rules of Suppositions in Formal Logic. *Studia Logica*, 1934. **1**:5–32.

[10] Fitch F. Symbolic Logic, An Introduction. The Ronald Press Company, 1952.

[11] Standefer S. Translations Between Gentzen-Prawitz and Jaśkowski-Fitch Natural Deduction Proofs. *Stud Logica*, 2019. **107**(6):1103–1134. doi:10.1007/s11225-018-9828-2.

[12] Zandleven I. A verifying program for Automath. In: Nederpelt R, Geuvers J, Vrijer, de R (eds.), Selected Papers on Automath, Studies in Logic, pp. 783–804. North-Holland Publishing Company, Netherlands. ISBN:0-444-89822-0, 1994 (originally 1973).

[13] Wiedijk F. A New Implementation of Automath. *J. Autom. Reason.*, 2002. **29**(3-4):365–387. doi: 10.1023/A:1021983302516.

[14] van Daalen D. A description of Automath and some aspects of its language theory. In: Nederpelt RP, Geuvers JH, Vrijer, de RC (eds.), Selected Papers on Automath, Studies in Logic, pp. 101–126. North-Holland Publishing Company, Netherlands. ISBN:0-444-89822-0, 1994 (originally 1973).

[15] Kamareddine F, Nederpelt R. A Useful lambda-Notation. *Theor. Comput. Sci.*, 1996. **155**(1):85–109. doi:10.1016/0304-3975(95)00101-8.

[16] Bertot Y, Castéran P. Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN:978-3-642-05880-6. doi:10.1007/978-3-662-07964-5.

[17] Barendregt HP. Lambda calculi with types. In: Abramski S, Gabbai D, Maiboum T (eds.), Handbook of Logic in Computer Science, pp. 117–309. Oxford University Press, Oxford, 1992.

[18] Coquand T, Huet GP. Constructions: A Higher Order Proof System for Mechanizing Mathematics. In: Buchberger B (ed.), EUROCAL '85, European Conference on Computer Algebra, Linz, Austria, April 1-3, 1985, Proceedings Volume 1: Invited Lectures, volume 203 of *Lecture Notes in Computer Science*. Springer, 1985 pp. 151–184. doi:10.1007/3-540-15983-5_13.

[19] Barendregt HP, Geuvers JH. Proof-Assistants Using Dependent Type Systems. In: Robinson JA, Voronkov A (eds.), Handbook of Automated Reasoning (in 2 volumes), pp. 1149–1238. Elsevier and MIT Press, 2001. doi:10.1016/b978-044450813-3/50020-5.

[20] Geuvers JH. Proof assistants: history, ideas and future. *Sadhana Journal*, 2009. **34**(1):3–25.

[21] Wiedijk F. The Seventeen Provers of the World, Foreword by Dana S. Scott, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006. doi:10.1007/11542384_1.

[22] Naumowicz A, Kornilowicz A. A Brief Overview of Mizar. In: Berghofer S, Nipkow T, Urban C, Wenzel M (eds.), Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009 pp. 67–72. doi:10.1007/978-3-642-03359-9_5.

[23] Gordon M, Melham T. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, 1993.

[24] Gordon M. From LCF to HOL: a short history. In: Plotkin GD, Stirling C, Tofte M (eds.), Proof, Language, and Interaction, Essays in Honour of Robin Milner. The MIT Press, 2000 pp. 169–186.

[25] Church A. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 1940. **5**:56–69.

[26] Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL – A Proof Assistant for Higher-Order Logic, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi:10.1007/ 3-540-45949-9.

[27] de Moura L, Ullrich S. The Lean 4 Theorem Prover and Programming Language. In: Platzer A, Sutcliffe G (eds.), CADE 28, 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings, volume 12699 of *Lecture Notes in Computer Science*. Springer, 2021 pp. 625–635. doi:10.1007/978-3-030-79876-5_37.

[28] Bove A, Dybjer P, Norell U. A Brief Overview of Agda – A Functional Language with Dependent Types. In: Berghofer S, Nipkow T, Urban C, Wenzel M (eds.), Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Germany, August 17-20, 2009. Proceedings, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009 pp. 73–78. doi:10.1007/978-3-642-03359-9_6.

[29] Nordström B, Peterson K, Smith JM. Programming in Martin-Löf's Type Theory : an introduction. Oxford Science Publications, 1990.