A Nice and Accurate Checker for the Mathematical Language Automath

Abstract

This is the manual of version 4.1 of *aut,* a free[1], fast and portable checker (entirely written in the programming language C) for the languages AUT-68 and AUT-QE. In order to make this paper readable on its own, a short overview of Automath is included. After the description of the checker some performance statistics are given. Then a number of issues are mentioned which were encountered while implementing the program and a few thoughts on Automath are presented.

1. Intro

1.1. Automath

Automath is an artificial language for writing mathematics which was designed by N.G. de Bruijn around 1967. In Automath mathematical proofs are encoded in such detail that checking the correctness of those proofs with the aid of a digital computer becomes feasible. In 1994 a compilation of a number of papers about Automath has appeared as [NGV94] which now is the standard reference about the subject. However, this manual contains a short introduction to Automath as well and therefore should be readable in its own right.

It might appear that a system like Automath is some kind of theorem prover. However, the focus with Automath is not so much on having the computer *generate* a formal mathematical proof as well as using the computer to *verify* the formalization of a proof, while such a formalization very well might have been 'coded' entirely by a human being. This distinction is similar to that between a system designed to write 'computer generated literature'[2] and a word processor. Or between a system to do 'automatic programming' and a compiler for a programming language.

Automath today is mainly interesting because of its historical value: it was the first system of its kind.[3] However, in the field of proof checking there are quite a number of projects that have descended from – or at least are closely related to – Automath, the so-called *type theoretical* proof checkers. Some of them are still being actively worked on: there are the Coq project at INRIA in France [DFHHMPPW93], the Lego system of Randi Pollack in Edinburgh [LP92] and the Alf/Half/CHalf project in Göteborg [S96].

There are quite a number of other systems that were designed for the verification of mathematical proofs: for instance there are the Boyer-Moore theorem prover[4] [BM88]

1. Although it is *not* in the public domain: I explicitely claim all rights to it.

2. The novel *Just This Once* [F93] claims to have been written by a computer. I don't expect it to be great literature.

3. Of course there already existed formal or semi-formal languages for representing mathematics before Automath. For example there's the language of Whitehead and Russell's Principia Mathematica [WR27] as well as Freudenthal's Lincos (a language to communicate mathematics – among other things – to potential extra-terrestials) [F60]. However Automath was the first language expressly designed to make it possible to check complete proofs with a computer.

Also Automath is later than 'computer algebra' systems like Reduce and what came to be Macsyma [DST93]: work on these programs already started in the early sixties. However this kind of system never was intended for real mathematics: they only know how to calculate and not how to prove.

and the Mizar system [R92]. However, there is a big difference between these systems and the 'type theoretical' proof checkers: those last ones turn out to have a much simpler and cleaner foundation. In that sense they are far to be prefered to the more complicated systems because the 'proofs' that are created for them probably will have a much longer life: the elegance of the semantics they're based on will make it possible to relate them to more recent systems. (For example, it turned out to be quite easy to 'clone' the Automath implementation from the seventies – the program we report on here – while it seems to be almost unthinkable to do something similar for, say, the Boyer-Moore system.)

While the type theoretical proof checkers have a very elegant basis, and hence may be expected to give a proper framework for the digitalization of a significant piece of mathematics, there is a second class of systems that have a similarly simple underpinning. Those are the *resolution* based systems, of which the Otter system [M93] is the best known. However proofs by means of resolution are quite foreign to the way a 'human' would prove something. This means that the resolution based systems primarily are useful as proof generators and not as proof checkers: and in this sense they're not really applicable to checking a translation of existing mathematical knowledge.

1.2. Landau & Jutting

The Automath project (which was funded by the Dutch national organisation for pure research, which was then called ZWO) ran from the late sixties to the middle seventies. Although there were quite a number of people involved with it, as with most efforts of this kind there were three 'main' people in the project, one for each main 'role' in such an undertaking. There were the *theoretician,* who of course was N.G. de Bruijn, secondly the *implementor* of the computer system that processed the language, I. Zandleven and finally the *user* of the system who wrote the 'big' case study: L.S. Jutting.[5]

In the Automath project a number of example texts were written, but only two large case studies were undertaken. These were (i) the translation by Jutting of Landau's Grundlagen and (ii) a treatment of real analysis by J. Zucker. However, this second text was never entered in a computer (I even think it only exists in handwriting) and was written in a dialect of Automath (AUT-Π) which never was implemented. Hence this second case study never really was finished, and so there only exists one real large

4. Of course the Boyer-Moore theorem prover seems to be a proof generator and not a proof checker. However, in [BM88] on page 179 we read:

> *The key role of the user of our system is guiding the theorem prover to proofs by the strategic selection of the sequence of theorems to prove and the proper formulation of those theorems.*

and on page 207:

> *Complicated proofs are broken down into lemmas or cases, and each case is proved in isolation* [...] *Thus, the theorem prover is kept 'on a short leash.' This is an attractive strategy for many reasons. First, the system is more predictable – less is expected of it and so it more often meets your expectations. Second,* [...] *Third, you'll spend less time trying to trick it into producing the entire proof 'automatically.'*

This suggest that the 'theorem prover' really only can do the 'low level' steps of a proof and that the rest of the proof is provided by the user: and hence that such a proof really is 'checked' by the system.

5. Likewise, in the Half project from Göteborg one can point out these same roles: in that project the theoretician is Thierry Coquand, the implementor is Dan Synek and the user is Jan Cederquist.

Automath text: the one by Jutting.

In the twenties Edmund Landau wrote a book (he claims he did it for the education of his daughter) called *Grundlagen der Analysis* ('foundations of analysis') [L30] about the 'elementary' properties of the various kinds of numbers: starting from the natural numbers he introduced the integers, then the fractions & rationals, the real numbers and finally the complex numbers. In this little book he showed more or less what was needed to prove that the reals are a complete ordered field. It consists of just a long list of 'Sätze' (propositions) together with their proofs and is unusually detailed, even for a mathematical text. And, despite what one might expect, the mathematical level of this book is far from completely trivial: a number of the proofs turn out to be quite complex or even subtle.

In the early seventies Jutting translated Landau's book to get his Ph.D. This translation was made in 'second generation' Automath. The original Automath was called AUT-68 (presumably because it was finalized in 1968), but after a false start where Jutting translated the first chapter of Landau's book into AUT-68 he switched to the newer AUT-QE (the QE stands for 'quasi expression': apart from being a bit more efficient when coding 'logic' the major difference between this language and its predecessor seems to be that it allows for axiom schemes).[6]

To give an impression of Landau's book and Jutting's translation, I'll present here one proposition (which was randomly chosen: number 137) and its translation. You are not expected to understand this (below we'll present a much simpler example of an Automath text with proper explanation), it's just meant to give the flavor both of Landau's original German text and of Jutting's translation. Incidentally the size of this fragment is about one thousandth of the whole Grundlagen.

So first here is Landau's version ([L30], page 75):

Satz 137: *Aus*

$$\xi > \eta,\ \zeta > \upsilon$$

folgt

$$\xi + \zeta > \eta + \upsilon.$$

Beweis: Nach Satz 134 ist

$$\xi + \zeta > \eta + \zeta$$

und

6. The 'third generation' of Automath was to be AUT-$\prod$ (which would include Zandleven's much more efficient AUT-SYNT and that was to have all other kinds of fancy stuff like 'telescopes' – incidentally the one thing the Automath book [NGV94] doesn't properly explain as it doesn't contain [B91] because it 'is very recent and can easily be tracked'), but which as already has been mentioned didn't materialise anymore.

Apart from these three 'generations' – AUT-68, AUT-QE and AUT-$\prod$ – there are a couple of other Automath 'flavors' among which the most interesting from a theoretical point of view is de Bruijn's AUT-$\Delta\Lambda$ (a more recent and slightly more subtle version of his AUT-SL) which we'll briefly encounter below in section 5.2.

$$\eta + \zeta = \zeta + \eta > \upsilon + \eta = \eta + \upsilon,$$

also

$$\xi + \zeta > \eta + \upsilon.$$

And here is Jutting's translation of it:[7]

```
@[ksi:cut][eta:cut][zeta:cut][upsilon:cut]
upsilon@[m:more(ksi,eta)][n:more(zeta,upsilon)]
+3137
t1:=satz134(ksi,eta,zeta,m):more(pl(ksi,zeta),pl(eta,zeta))
t2:=ismore12(pl(zeta,eta),pl(eta,zeta),pl(upsilon,eta),pl(eta,upsilon),compl(zeta,eta),
    compl(upsilon,eta),satz134(zeta,upsilon,eta,n)):more(pl(eta,zeta),pl(eta,upsilon))
-3137
satz137:=trmore(pl(ksi,zeta),pl(eta,zeta),pl(eta,upsilon),t1".3137",t2".3137"):
   more(pl(ksi,zeta),pl(eta,upsilon))
upsilon@[l:less(ksi,eta)][k:less(zeta,upsilon)]
satz137a:=satz121(pl(eta,upsilon),pl(ksi,zeta),satz137(eta,ksi,upsilon,zeta,satz122(ksi,eta,
    l),satz122(zeta,upsilon,k))):less(pl(ksi,zeta),pl(eta,upsilon))
```

This is literally what's in the file, apart from the fact that in order to make the structure a bit more clear I made the parts of the text giving the 'statement' of the proposition a bit larger than the parts giving the 'proof'. Also, as will be clear Jutting proves the proposition in two different directions (he doesn't only give the 'greater than' case but also the 'less than' one), apparently for later convenience.

A few words about the relation between the sizes of both the German, 'natural language', version and the Automath version of the Grundlagen.[8] The book consists of 144 pages and a TeX version of it takes about 189 kilobytes. The AUT-QE translation[9] is 736 kilobytes: this means it's about 3.9 times as large as the TeX coding of the original. However, this factor is not really meaningful as it depends quite a bit on the way the two versions were formatted: how macros were used in the TeX version, how much whitespace was present, etc. So a more interesting ratio is that between *compressed*

7. Actually, the first line which is italicized is *not* part of the translation of proposition 137. Instead it consists of pieces of lines scattered throughout earlier parts of the text.

8. A claim in the Automath project was that ([NGV94], A.5, page 160):

> *A very important thing that can be concluded from all writing experiments is the* constancy of 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 contant: 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.*

Of course this is qualitative talk and it really is interesting to have quantitative data about this, for example to be able to compare various languages for their 'efficiency of abbreviation'. It turns out that in the case of the translation of the Grundlagen into AUT-QE this loss factor is slightly less than *four*.

9. More precisely: the 'main branch' of the translation – Jutting made two completely different translations of chapter 4 (both having about the same size).

versions of the two texts (we used gnu's gzip utility to compress them), because this will effectively hide that kind of dependence. Surprisingly enough, both files compress extremely well to respectively 42 and 155 kilobytes and those two sizes have approximately *the same* ratio as the original ones. To be precise, when compressed the Automath version is 3.7 times as large as the German one.

1.3. Damage

In 1992, I received from Jutting (who was introduced to me by my friend Hans Mulder who was at the time sharing a room with him) a number of 5.25 inch 'soft' floppy disks, on which presumably was the text of the Grundlagen translation. He told me that the files on those floppies had been transfered a couple of times from computer to computer in the past, and that during that process they had been 'damaged': pieces of the files now were missing. Also one file was present in two copies on the disks: presumably one contained some 'improvement' over the other.

It turned out that Jutting was right. A simple yacc grammar quickly pointed out a number of places where randomly a small part of a line was missing.[10] So, with the aid of printouts of the original version of the Grundlagen text, I started to repair the files.[11] The two 'versions' of that one file turned out to differ in only one line: it looked like someone already had been repairing one of the damaged places.[12]

The damage to Jutting's files were the main motivation for me to write a working Automath checker: that was the only way I could be certain I had repaired the files correctly. I might have used an existing checker if one still had existed but that turned out not to be the case. Before, it seems there had existed two checkers: Zandleven's, which was used by Jutting when he was working on his thesis, and a 'second' checker on which Jutting reports ([J79], page 33):

> *Meanwhile an entirely new program has been implemented by I. Zandleven and A. Kornaat.* […] *In April 1977 the whole Landau book was checked by this program, and thus the correctness of the AUT-QE text is now mechanically established.*

However both checkers apparently were written in a computer language that is no longer current and so they are no longer operational.

Of course the other reason that I wrote an Automath checker was that I really wanted to experiment with the language.

---

10. Specifically there were 15 such places, with

    14 + 15 + 33 + 35 + 32 + 59 + 36 + *35* + 27 + 53 + *64* + *33* + 35 + *21* + *42* = 534

bytes missing (and the pieces that are italicized contained an end-of-line byte, so five lines had been joined to the following one.) That's on average about 32 bytes about every 64 kilobytes.

11. When the full checker was finished it turned out that I had made no typing errors in these 'repairments'; but I had missed one place where the file was damaged.

12. I originally expected that this would be about two 'different' translations of the same passage, but this was not the case: the 'shorter' version was just missing 48 bytes of the 'larger' version and was not even syntactically correct Automath.

2. Example

2.1. A very small proof

As a short introduction to Automath, and as a vehicle to show some language issues, I present here a very small example of an Automath text.[13] It gives the proof that the 'double negation rule' implies 'contradiction elimination'.

In propositional logic – natural deduction style – the logical connectives have 'introduction' and corresponding 'elimination' rules. This is all quite pleasant, apart from the fact that it doesn't give us the common, classical logic but instead a constructive one. What's missing is the 'double negation rule' which says that from $\neg\neg p$ we may derive p. However, once we add this rule the elimination rule for the contradiction, which says that from $\bot$ we may derive any p, becomes superfluous. This is what our example proves.

The proof is really quite simple. If we have a contradiction, we have *any* negation (for by definition the negation of a proposition is the implication of a contradiction) so in particular we have any double negation (a double negation surely is a negation). And therefore by the double negation rule we have any proposition.

2.2. Logic type theoretically

In Automath everything either is an *object* or a *type*, which themselves may depend on other objects and types. Each object has a type, while all types are of the unique *kind* called TYPE.[14] The typing relation is written with a colon, so if x is an object of type t we write $x{:}t$, while if t is a type we write $t{:}$TYPE.

So here's how to do propositional logic in this framework.[15] There is a type called *Prop* for the propositions and for each object p of type *Prop* there is a type *Proof(p)* for the 'proofs' of p. These are the sole type constructors that will be needed for logic. Then there are the contradiction *con* and for each p and q of type *Prop* the implication *imp(p,q)* which are both objects of type *Prop*. And finally for each introduction or elimination rule there's a corresponding object that is of the appropriate *Proof* type. For instance implication elimination ('modus ponens') is given by *impelim(p,q,tp,timp)* which is an object of type *Proof(q)* and where the parameters p and q both have type *Prop*, *tp* has type *Proof(p)* and *timp* has type *Proof(imp(p,q))*.

Now if we write $f(x_1{:}t_1,\ x_2{:}t_2,\ \ldots,\ x_n{:}t_n) : t$ when we have an f of type t that

13. Actually it is not a very *simple* example, as one needs to know about natural deduction proofs to understand it. But it is very *nice.*

14. In Automath parlance the objects are called 3-expressions, the types 2-expressions and the kinds 1-expressions. Often there is a second kind as well which is called PROP. In type theory the kind TYPE is usually written as *.

15. The way to encode logic that's presented here is not the efficient way of AUT-QE which is used in Jutting's text. Instead, it's the slightly more involved way of AUT-68. This means that the 'propositions as types' pardigm is not adhered to. On the other hand, in the way things are done here it seems more apparent that 'proofs are objects'. For a discussion of how to add the 'p-fragment' and how the analogy between objects/types and proofs/propositions works, see [NGV94], A.3 (in particular, the table in 4.5 on page 115 is really enlightening.)

depends on $x_1, x_2, \ldots, x_n$ of types $t_1, t_2, \ldots, t_n$, then using this format the implication elimination constructor becomes:

impelim(p:Prop, q:Prop, tp:Proof(p), timp:Proof(imp(p,q))): Proof(q)

Using this notation, the notions from the example are given by:

Prop: TYPE
Proof(p:Prop): TYPE
con: Prop
imp(p:Prop, q:Prop): Prop
not(p:Prop): Prop
impintro(p:Prop, q:Prop, tq:[tp,Proof(p)]Proof(q)): Proof(imp(p,q))
impelim(p:Prop, q:Prop, tp:Proof(p), timp:Proof(imp(p,q))): Proof(q)
notnotelim(p:Prop, tnn:Proof(not(not(p)))): Proof(p)
conelim(p:Prop, tcon:Proof(con)): Proof(p)

This list contains an instance of what's still missing from our description of type theory: lambda abstraction and function application. When writing down expressions we not only build them from notions that have been introduced before, but we too may write and apply functions the way it is done in lambda calculus. The Automath notation for this is slightly idiosyncratic: a function that assigns some *e* to an argument *x* of type *t*, is written in Automath as [*x*,*t*]*e* instead of the more customary $\lambda_x\varepsilon$. So, the *tq* argument of *impintro* in the enumeration above is actually a function that gives for each object *tp* of type *Proof(p)* a value of type *Proof(q)*.[16] Function application is written differently in Automath also: if *f* is a function object and *x* an argument instead of writing *f(x)* or *f x* in Automath one writes <*x*>*f*.

This actually is all there is to AUT-68. There is quite a bit 'housekeeping' machinery there too which hasn't been described yet, but that all is non-essential syntactic sugaring.

So what's the proof of the example? Because all we've given thus far is a list of the various things to be defined. Well actually all but two of the notions in that list are constructors, what Automath calls *primitive notions* (one has to start somewhere after all, so it is not strange that in such a short example most of the things are axioms.) The only two notions that are *not* primitive are defined in the following way:

not(p) $=_{\text{def}}$ imp(p,con)
conelim(p,tcon) $=_{\text{def}}$ notnotelim(p,impintro(not(p),con,[tnot,Proof(not(p))]tcon))

That right hand side of the second line is the proof.

---

16. The fact that if *e* has type *t* the function [*x*] *e* has type [*x*]*t* makes one wonder whether the straight brackets are muddling the distinction between lambda abstraction and product types, or whether maybe that distinction is not so clear in the first place, and also what the *kind* of [*x*]*t* should be. This is an interesting topic that we will not go into here.

2.3. Various formats

Of course the text in the previous subsection is not syntactically correct Automath: it's just meant to make the structure of the thing clear. Here is how the same proof would be written in old-fashioned AUT-68:

```
      * Prop          :=      PN                  ; TYPE
      * p             :=      ---                 ; Prop
    p * Proof         :=      PN                  ; TYPE
      * con           :=      PN                  ; Prop
    p * q             :=      ---                 ; Prop
    q * imp           :=      PN                  ; Prop
    p * not           :=  imp(p,con)              ; Prop
    q * tq            :=      ---                 ; [tp,Proof(p)]Proof(q)
   tq * impintro      :=      PN                  ; Proof(imp(p,q))
    q * tp            :=      ---                 ; Proof(p)
   tp * timp          :=      ---                 ; Proof(imp(p,q))
 timp * impelim       :=      PN                  ; Proof(q)
    p * tnn           :=      ---                 ; Proof(not(not(p)))
  tnn * notnotelim    :=      PN                  ; Proof(p)
    p * tcon          :=      ---                 ; Proof(con)
 tcon * conelim       :=  notnotelim(impintro(not,con,[tnot,Proof(not)]tcon))
                                                  ; Proof(p)
```

Some explanations: The main difference between this representation and the one we gave before is that the parameters have been taken out from between their brackets and turned into 'first class citizens'. They now are more or less like the other notions, except that their 'definition' part is a 'long dash' (simulated by a triple hyphen) to indicate that it's a parameter. Also, each parameter 'extends' a chain of other parameters which is indicated before the asterisk. For instance the parameter *timp* extends the chain *p,q,tp*. This actually is very efficient: often the parameters of various notions will have large common prefixes, and this way those won't be to be repeated all the time. There is another 'shorthand' that complements this practice: when a notion is invoked an initial segment of the parameters may be left out when they are the same as the ones at the time of definition. For instance one may say *not* instead of *not(p)* like in the last line, because the parameter of *not* turned out to be *p* already. Together this means that in a huge context of parameters one may name some complicated expression and then refer to that name without having to worry about the parameters at all.

The way we wrote this Automath text is not at all like the content of the files I received from Jutting. Here is how it would have been coded there:

```
@Prop:='prim':'type'
[p:Prop]
```

```
Proof:='prim':'type'
@con:='prim':Prop
p@[q:Prop]
imp:='prim':Prop
p@not:=imp(p,con):Prop
q@[tq:[tp:Proof(p)]Proof(q)]
impintro:='prim':Proof(imp(p,q))
q@[tp:Proof(p)][timp:Proof(imp(p,q))]
impelim:='prim':Proof(q)
p@[tnn:Proof(not(not(p)))]
notnotelim:='prim':Proof(p)
p@[tcon:Proof(con)]
conelim:=notnotelim(impintro(not,con,[tnot:Proof(not)]tcon)):Proof(p)
```

As will be apparent there are some lexical changes here: the stars have become ‘at’ signs and some semicolons and comma’s have mutated into colons. Also, the ‘keywords’ PN and TYPE have been spelled differently. But more importantly, the ‘context part’ of the lines (the label in front of the * that shows on what chain of parameters the line builds) is treated quite differently. It turns out that most of the time after a parameter line the context needed is just that parameter, and after a non-parameter line it hasn’t changed. Apparently in this dialect of Automath, in such case the context part may be omitted. This means that in this ‘optional context part’ variant of Automath, in a sense the lines have been ‘cut in half’ (in fact: after the *): the first half can be used to set the current context and the second half defines the notion or the parameter, and both kind of halves may now be mixed freely.

So apparently it turns out there are a fairly ‘loose’ Automaths around. In that case I think I should be allowed my own dialect as well. Here it is:

```
 @      Prop:TYPE =PRIM
    [p:Prop]
        Proof:TYPE =PRIM
 @      con:Prop =PRIM
p@  [q:Prop]
        imp:Prop =PRIM
p@      not:Prop =imp(p,con)
q@  [tq:[tp,Proof(p)]Proof(q)]
        impintro:Proof(imp(p,q)) =PRIM
q@  [tp:Proof(p)] [timp:Proof(imp(p,q))]
        impelim:Proof(q) =PRIM
p@  [tnn:Proof(not(not(p)))]
        notnotelim:Proof(p) =PRIM
```

```
p@  [tcon:Proof(con)]
        conelim:Proof(p)
        =notnotelim(impintro(not,con,[tnot,Proof(not)]tcon))
```

This is almost identical to the previous version, apart for some added whitespace and yet another lexical change: the ‘assigned’ has turned into a single equal-sign. But the real change here is that I interchanged the ‘type’ and ‘definition’ parts of the line and put the first one in front of the second. This makes the text closer to the list of notions from section 2.2 (just ignore everything after the equal signs), and it’s also more like standard mathematical practice: when proving a proposition the statement of the proposition will now be in front of the proof.

Of course the Automath checker which is described here happily accepts all three variants of the text.

3. Practical

3.1. Where to get it & how to install it

The Automath checker *aut* can be obtained by way of the world wide web at <http://www.phil.ruu.nl/~freek/aut-4.1.tar.gz>. A page giving some information about the *aut* project can be found at <http://www.phil.ruu.nl/~freek/aut.html>.

After obtaining the package it has to be installed. To do this one should run *gunzip* and *tar* to unpack it and then *make* to build it. These are the commands to do this on a Unix system:

```
gunzip aut-4.1.tar.gz
tar xf aut-4.1.tar
cd aut-4.1
make
make test
```

The last command will run the Grundlagen as a shallow test. If all went well, after all this the directory *aut-4.1* will contain a binary called *aut* which is all that need to be retained (it doesn’t need any other file) and which if desired can be copied to a more public location.

There also is a precompiled MPW tool version for use with the Macintosh Programmer’s Workshop [M87].[17] This tool is binhex encoded in the file *aut.hqx* and so should first be unpacked with a tool to unpack binhex like Aladdin’s StuffIt Expander. It is a ‘fat binary’ which means that it will both run on the ‘old’ 680X0 Macintoshes as well as on the ‘new’ PowerPC models, and that in the second case it will execute at ‘native’ speed.

17. Of course if one is only going to use the MPW tool it doesn’t make sense to do the *make* commands.

3.2. Syntax of the language

Here is a context free grammar of the language accepted by the checker, derived directly from the lex and yacc sources of the parser (as the syntax is fairly loose, quite a number of variations in these rules actually mean exactly the same):

```
text ← | text item | text ';'
item ← '+' IDENT | '+' '*' IDENT | '-' IDENT | '--'
    | STAR | sym STAR
    | IDENT ASSIGN EB SEMI expr | IDENT SEMI expr ASSIGN EB
                | '[' IDENT COMMA expr ']'
    | IDENT ASSIGN PN SEMI expr | IDENT SEMI expr ASSIGN PN
    | IDENT ASSIGN noexpand expr SEMI expr
                | IDENT SEMI expr ASSIGN noexpand expr
noexpand ← | '~'
expr ← 'TYPE' | ''type'' | 'PROP' | ''prop''
    | sym | sym '(' ')' | sym '(' exprlist ')'
    | '<' expr '>' expr | '[' IDENT COMMA expr ']' expr
exprlist ← expr | exprlist ',' expr
sym ← IDENT | IDENT '"' identlist '"'
identlist ← | IDENT | identlist HYPHEN IDENT
EB ← 'EB' | ''eb'' | '---'
PN ← 'PN' | ''pn'' | 'PRIM' | ''prim'' | '???'
STAR ← '*' | '@'
ASSIGN ← ':=' | '='
SEMI ← ''E'' | 'E' | ';' | ':'
COMMA ← ',' | ':'
HYPHEN ← '-' | '.'
```

An *IDENT* (for 'identifier') is an arbitrary sequence (but not equal to any keyword) of characters from the set of letters, digits, the single quote, the backquote, the underscore and the backspace character (that's what the E keyword is: an underscore, then a backspace and finally a capital 'E'.) Whitespace separates lexical items and is, well, about any kind of whitespace. Comments are either from a '#' or '%' character until the end of the line (which makes it possible to start an Automath file with the string '`#!/usr/local/bin/aut`') or consists of text enclosed between '{' and '}' characters (with nesting being taken into account).[18]

The first line of the *item* rule and the *identlist* in the *sym* together are Automath's 'paragraph system'[19] which gives a rather crude kind of modular structure to an

18. So a large piece of text can be 'commented out' by putting it between braces: however this doesn't work quite as it should when there are mismatched braces in end of line comments. This is too bad.

19. Which is a misnomer really. The Dutch word 'paragraaf' translates into English as 'section', while the Dutch word for 'paragraph' is 'alinea'. So it really should have been called 'sectioning system'.

Automath text by structuring its name space in a way similar to Unix directories. Roughly speaking a line like '+3137' enters a 'subparagraph' called 3137 and a line '-3137' (or just '--') leaves it again, while a line '+*3137' means that a paragraph which already has been visited is entered again.

There are two ways to use the paragraphs when referring to something. An expression like 't1"-3137"' is about the identifier 't1' from a subparagraph '3137' which should be a direct descendent of the current paragraph. An expression like 't1"3137"' – so without the leading hyphen/period – refers to any paragraph '3137' containing the current one. Finally one can create 'paths' of references to paragraphs: 'th2"l-r-imp"' is about the notion 'th2' from paragraph 'imp' inside paragraph 'r' inside paragraph 'l'. For a more precise description of all this see [J79], appendix 2, pages 78-83.

The *noexpand* tilde means that a definition may not be expanded (that is: used in delta reductions) when checking. The idea of this is that this can be used to speed up the checking process by have it avoid unnecessary work.[20]

3.3. Invocation & flags

The way the checker is run is standard for Unix: arguments that start with a hyphen are *flags* that indicate special options to the program. All other arguments are names of files (possibly with an implicit suffix *.aut* omitted) which are checked as one input stream in the order in which they appear on the command line. If no file arguments are present the program will try to read the Automath text from the terminal ('standard input'). For instance after the command

```
aut -QZ grundlagen
```

the file *grundlagen.aut* will be checked with the flags -Q (telling the checker to expect AUT-QE) and -Z (report all statistics) turned on.

Here is a list of all the flags that the *aut* checker knows about together with their meaning:

-Q AUT-*Q*E – the language of the Grundlagen: this is equivalent to '-acopqs'

-a *a*llow *a*bstractions of degree one – so expressions like *[x,sigma]TYPE* are acceptable

-c put *c*ontext parameters in front of other notions – see section 4.1

-o *o*ld style parameters: disallow explicit paragraph references – paragraph qualifications are disallowed in parameter instances

-p allow *p*rops – TYPE and PROP are both degree one primitives (else PROP and 'prop' behave like ordinary identifiers)

-q allow *q*uasi expressions – use the AUT-QE typing system: 'parametrised types' of the kind *[x,sigma]TYPE* are allowed where something like a *TYPE* is expected ('product types are types too'[21])

20. However, putting in tildes in lots of definitions seems rather tedious.

-s disallow paragraph reopeners without a *s*tar – the asterisk which indicates that one reenters a paragraph that's already known is compulsory

-Z summarize everything – this is equivalent to '-dlmrv'

-d summarize *d*uration – print the number of seconds the program executed
-l summarize number of *l*ines – print the number of various kinds of 'item's (PN, definitional and EB 'lines') in the text
-m summarize *m*emory usage – print the amount of heap memory used in kilobytes
-r summarize number of *r*eductions – print the number of various reductions (beta, delta, eta) needed to check type correctness
-v summarize *v*ersion – print the version number of the *aut* checker and a copyright notice
-z summarize to stdout instead of stderr – normally the 'summarize' flags print to 'standard error': use 'standard output' instead

-b omit lambdas when calculating categories of degree one – an expression like *[x,sigma]tau* will have category TYPE instead of *[x,sigma]*TYPE
-e disallow *e*ta reductions – don't try eta reductions when trying to prove expressions equal: however the checker is built in such a way that if a check without eta reductions is possible it will always be found first
-f *f*ull check: ignore squiggles – even try expanding notions that have been marked *noexpand*
-i print *i*nternal addresses of lambda expressions – this flag is intended for debugging the checker: when an abstraction or variable is printed the address of its location in memory is printed with it
-y print everything – pretty print the Automath text back to standard output

The next four flags all expect a numeric argument *n* in the next command line parameter, which may refer to a line number in the Automath input stream.

-k *n* how many implicit parameters to *k*eep – controls how many arguments are given when an expression is printed (0: only print the parameters that are really required, 1: print the parameters that were present in the original, 2: print all parameters)
-n *n* limit to *n* reductions per typecheck – intended to keep the checker from filling all memory when a check fails: when the given limit is exceeded a tentative type error is given and checking continues with the next item
-t *n* *t*race reductions – print the reductions tried when working on the type check of line *n* (or if *n* is 0: trace everything)

21. Or: 'why not do untyped lambda calculus at the bottom and see whether we can still escape the paradoxes?'

-x *n* print e*x*cerpt – print all those lines from the input needed for the statement of line *n* (if *n* is 0: print an excerpt for the primitive notions present)

4. Background

4.1. Language subtleties

There were a few issues with the interpretation of Automath which I hadn't anticipated at all when starting work on the checker. I'll briefly present them here.

The most interesting is related to the options '-c' and '-o' of the checker. Because of the symmetry in the 'original' Automath between parameter lines and definitional lines (the only difference being that for parameters the 'body' is 'empty', that is, '---'), I would have expected both kinds of items to 'live in the same name space'. However, in the Landau text by Jutting this appeared not to be the case. Consider the following Automath fragment:

```
    * t          :=    PN          ; TYPE
    * x          :=    ---         ; t
x   * y          :=    ---         ; t
    * x          :=    PN          ; t
y   * f          :=  x             ; t
```

Now what is the *x* in the last line refering to? Is it the parameter from line 2 or is it the constant from line 4? Or maybe it's illegal to 'override' a parameter by a constant this way?

I had at least expected that the *x* in the last line would *not* refer to the first one, as clearly the it is in the 'scope' of the second line. However, it turned out that Jutting's text contained lines that showed that a unqualified name was *first* tried to be interpreted as one of the parameters and only *then* as referring back to some defined notion. Clearly in the original Automath checker they lived in different name spaces, with the parameters being given 'preference'. And if one closely reads [J79], page 82, well, maybe it says so there.

Actually it took quite some effort to make the checker behave sufficiently different from my original interpretation that it was able to check the Grundlagen: and even in that mode it doesn't really put the parameters in a different name space (actually I don't think it wouldn't be very efficient). The way the checker works now is that (a) the example above is legal anyhow, (b) without the '-c' flag the *x* refers to the constant from line 4 and (c) with the '-c' flag it refers to the parameter from line 2. This is implemented by, if the '-c' flag is on, opening a new 'block' with the current parameters before starting the interpretation of the identifiers in the body and category parts of the line and closing it again afterwards. Of course, most of the time this doesn't change anything because in practice parameters hardly ever will be 'masked' in this manner.

The other issue that I hadn't anticipated was the question what should happen to the 'implicit' context when changing paragraphs. Consider this:

```
        @ t             :=      PN              ; TYPE
          x             :=      ---             ; t
+p
          y             :=      ---             ; t
-p
          f             :=      PN              ; t
        @ z             :=      ---             ; t
+*p
          g             :=      PN              ; t
-p
```

What are the arguments of *f*? Is it the context *x, y* (because *y* was the last parameter introduced before the *f*) or is it *x* (because the *y* was introduced 'inside' paragraph *p* and so is no longer in scope after that paragraph has been left)? Similarly is the context of *g* the *x, y* that is 'reactivated' when *p* was reopened or is it the *z* from the previous line? (Note that this kind of ambiguity only occurs because the 'context indicator' may be implicit.)

Well, of course it doesn't matter much what choice one makes for the answer to these questions. The interpretation that *aut* currently implements is that it's *f(x)* and *g(x,y)*. This second choice was needed to correctly interpret the Grundlagen but the first one was not, because in Jutting's files the context after a paragraph closer is never left implicit. However, these two choices seem consistent with each other.

4.2. Memory management

Dynamic memory allocation is a pain: having to free blocks of memory manually takes quite a bit of programming effort and it's easy to make errors in it. So in version 1 of the *aut* checker I decided not to do it, but to use an automatic garbage collector instead (the Boehm-Weiser *gc* 'conservative' garbage collection library, which can be used with traditional programming languages like C and C++ ([BW88]; on the web there might be a page about it at <`http://reality.sgi.com/employees/boehm_mti/gc.html`>).[22] However this had some disadvantages: it made it harder to debug the program (it was less predictable what would happen during a run and if a crash occurred inside the garbage collector it was not very easy to find out what the checker had been doing at the time), it made it rather difficult to run the program on ancient computers like Apple Macintoshes, it made the program less portable (for instance porting it to a new flavor of Unix turned out not to be *entirely* trivial) and it made the system quite a bit larger and not self contained as well.

So in version 4 of *aut* I took a different approach. Because the checker is a very simple one-pass system, it turned out to be easy to write a trivial memory manager that's simple and fast. The idea is that either we are parsing an expression from the Automath source and building a data structure in memory which will never go away anymore or we

22. Coincidentally this same garbage collector was used by Dan Synek in his CHalf system.

are doing a type check and after that check succeeded or failed we can 'throw away' all memory that we used for it. So the memory manager has just to behave like a stack. New blocks are allocated by breaking pieces from one end of a large pool of memory[23] with a pointer pointing to where we have come thus far. When we start a 'type check' and know that at the end of it all extra memory that we are going to allocate won't be needed anymore, we just have to remember the location of this pointer and after the check is over we put it back where it was (we free the memory by 'popping' it all at once.)

This means that the memory usage of the checker will follow some kind of 'saw toothed' pattern: it will grow a bit and be put back but not all the way, after which it will grow again, be put back again somewhat, etc. This means that the checker will approximately need as much memory as is needed to store the parsed Automath and to do the check of one (the last) line (as the last 'tooth' of the saw is the highest). Also, using this scheme memory will not fragment at all.

These expectations were confirmed by experiment. It turned out that this allocation strategy was implemented very easily and appeared to work quite well. Of course this scheme was only possible because of the trivial, one-pass structure of the program.

4.3. Yet another way to do alpha conversion

There are two common ways to represent lambda terms inside a computer: one can use named variables and then there's the nameless representation of de Bruijn. In the first method each lambda is labeled with some identifier and a variable that refers to a lambda is labeled too: with that same identifier. However, this system has the disadvantage that sometimes one needs to rename lambdas and variables in order to have everything work out right: this is the so-called 'alpha conversion'.

The other method, the one by de Bruijn (as described in [NGV94], C.2), doesn't label variables using an identifier but instead marks them with a natural *number*. This number indicates how many nested lambda abstractions should be skipped to get to the lambda the variable is refering to. The main disadvantage of this method is that when a term is substituted for a variable a lot of numbers in the substituted term will change, which means that that term can't be reused so that continuously new copies of such terms have to be built.

The *aut* checker uses a slight variation on the first method which I rather like. The main difference with the customary method is that effectively *each time* when a term is built all lambdas are given fresh names (this implies that two differing terms will always have different names for their outermost lambdas.) Furthermore for implementation efficiency instead of using an alphanumeric strings for those names the *memory address* of the memory block that's representing the lambda node in the term tree is used.

The procedure for doing a substitution for a variable in an expression using this representation is slightly subtle but not difficult. The implementation in the checker is given by the functions *substvar* and *substargs* in the source file *type.c*, totaling 70 lines of C code (mostly dealing with the various cases that occur when recursing over the

23. Actually it's slightly more complicated because for maximal flexibility this large 'pool' of memory is itself divided into largish blocks as well, blocks that will have to be managed too.

original term where the substitution occured in.)

Just like with the nameless method one still may have an alphanumerical identifier associated with the lambda nodes in order to be able to print terms to the user in a humanly readable way. Of course, because those identifiers aren't really used to identify the structure of the term it doesn't matter that these labels may conflict between nested lambdas. Still it is not nice for a user to see a term printed in which it isn't clear to what lambda a variable is refering (in fact, just to have a way of being *sure* of what the real structure of a term is, the checker has the '-i' flag.) Therefore the checker *does* contain a routine to do alpha conversion after all: but it is only invoked when a term has to be printed (which is slow anyway, so in which case the additional delay because of the alpha conversion will not be too important.) This routine scans the term and whenever it finds significant conflicts between nested lambdas it renames the outer lambda by putting quotes after its identifier until the problem goes away. Because in the checker a term is never changed once it has been built (in Lisp terms: there never occur operations like *replaca* and *replacd)* and because the *outside* lambda is changed this renaming will never cause a variable to be renamed in a context where it wouldn't have been necessary.

5. Outlook

5.1. Performance

When the *aut* checker is finished checking the Grundlagen with the '-Z' flag turned on, it will print something like:

```
8583 beta reductions, 20004 delta reductions, 2 eta reductions
32 + 6878 = 6910 definitions, 4297 + 6910 = 11207 lines
94 blocks = 3006 kilobytes
171 seconds = 2 minutes 51 seconds
aut 4.1, (c) 1997 by satan software
```

Of course the number of seconds will depend on the specific run and on the system it has been executing on. This example output was from a 20 MHz 68030 (a Mac LC III) but on a modern PowerMac with a 200 MHz 604e and a 256 kB of level 2 cache the same check takes between four and five seconds. Because this last machine has a performance of about 7 SPECint95, it seems that the check of the Grundlagen text takes of the order of 30 SPECint95·s, which means that the checker runs at a speed of about 400 lines/(SPECint95·s) or about 25 kB/(SPECint95·s) of non-whitespace Automath.[24]

The time *aut* takes can be broken down into three parts: the time needed to scan and parse the input file (implemented using flex and yacc: about 28%), the time used to build the internal representation of the Automath text (about 18%) and the time it takes to actually do the type check (about 54%). The *aut* checker is also continuously inspecting the sanity of the parameters of its routines and of its data structures (by using something like the C 'assert' macro) which needlessly slows it down by about 4% (but which is

24. Of course this kind of benchmark arithmetic is *completely* inappropriate: but it gives one *some* estimate, within a factor of two or so, of how long a check will be going to take on a given machine.

really indispensible for debugging.)

5.2. Automath as assembly

In [NGV94], B.2 de Bruijn explains that expanding abbreviations is essentially the same thing as beta reduction and hence that one can encode an Automath 'book' as a 'single line'. For this purpose in [NGV94], B.7 he defines the language AUT-ΔΛ, in which a representation of a full mathematical proof is built from just three primitives: the 'kind' called TYPE, function abstraction *[t]e*, and function application *<x>f*. As an example of this, when we translate the example from section 2 into AUT-ΔΛ and write it using nameless encoding, we get something like:[25]

```
[*][[]*][1][[2][3]4]([3]<2><>1)[[3]4][[4][5][[<1>5]<1>6]<<1>
<2>4>6][[5][6][<1>6][<<1><2>5>7]<2>8][[6][<<<>3>3>6]<1>7]([7
][<6>7]<<[<<1>5>8]1><7><<1>5>4><1>2)[[7][<6>7]<1>8]
```

Here is a fragment (to be precise: the definition of *not*) of the AUT-68 original of this translation, to enable the reader to see the styles of the 'single line' translation and its original next to each other:

```
    * Prop        :=     PN          ; TYPE
    * p           :=     ---         ; Prop
      ...
    * con         :=     PN          ; Prop
  p * q           :=     ---         ; Prop
  q * imp         :=     PN          ; Prop
  p * not         :=  imp(p,con)     ; Prop
      ...
```

Now let's compare this to three kinds of computer program.

The most primitive way to write a program is to enter it directly in machine code (just toggle it in on the front panel of the console of the machine):

```
4E56 0000 486D FAC4 4EAD 014A 4E5E 4E75
846D 6169 6E00 0000 2057 42A7 486D FC96
486D FC92 486D FC8E 2F08 4EBA 01C8 72FF
```

25. A few remarks on the specific representation that's used here. De Bruijn writes τ for TYPE, but we use *. In [NGV94], B.7 on page 337 a distinction is made between strong and weak function application: we write *<x>f* for strong and *(x)f* for weak. De Bruijn 'counts' abstractions differently from us because he starts counting at one and we start counting at zero: this means that he denotes $\lambda_x\lambda_y x$ as [τ][τ]2 but we write it as [*][*]1. Finally, for aesthetic purposes we represent 0 with zero digits, that is, using the empty string.

```
B041 672E 588F 4FEF 0010 486D FEB6 4EBA
…
```

Then there's assembly, which is rather close to machine code actually, being the same thing, only in a more readable notation:

```
main        LINK    A6,#0
            PEA     hello(A5)
            JSR     printf(A5)
            UNLK    A6
            RTS
…
```

Now to me the resemblance between the two kinds of Automath on the one hand and these two kinds of computer program on the other hand is remarkable. In particular the similarity between the AUT-68 text and the assembly is striking: they're both line based texts, the lines are divided into various 'fields', those fields are written in a columnar format, etc.

But for a computer program we have a third way to write it, which is in a 'high level' programming language:

```
main() { printf("hello, world\n"); }
```

The point I'm trying to make here is that it might be advantageous to go look for some kind of new approach that does for proof checking what high level languages did for computer programming.

The transition between assembly and high level programming languages was made in the late fifties with FORTRAN, which – the name FORmula TRANslator already makes this clear – mainly differs from assembly in that it knows how to evaluate artithmetical expressions that have been written algebraically, without the programmer having to decompose those into elementary steps.

I think that the direction to look for a more 'high level' proof checking language should be a similar one. When reasoning mathematically, people are always – most of the time unconsciously – making identifications between things that are really 'the same'. For instance, when reasoning about topological spaces one really doesn't remember all the time that what one *really* is talking about is specific representations of such spaces with homeomorphism being the relevant equivalence.

Or, after having proved a number of equivalent characterizations of some property and having given that property a name, one doesn't worry anymore which characterization was the 'definition' and which ones were 'derived': it's 'all the same' after all, so it doesn't matter. This means that mathematicians are constantly manipulating with *congruences*: equivalence relations that are conserved under some interesting class of

operations. And they're so good at it that most of the time they're unconscious of it: and so when digitizing a proof they shouldn't be bothered with it either.

Another argument to be interested in congruences is that when we look at the way computer algebra systems and theorem provers are implemented, quite often their core engine is a *term rewriter.* And term rewriting typically is about congruences too: one subsequently replaces subexpressions of a term by something that's congruent to it. So if we want to be able to integrate computer algebra systems in our proof checkers or if we want to tap into the power of theorem provers, looking at congruences also seems to be the way to go.

So what I think is needed is something that might be called CONPROOF[26]: an Automath-like system that is able to automatically insert CONgruence PROOFs. In such a system one would write a chain of congruences and the system should be able to figure out what known congruence lemma's will need to be applied to establish those steps, without the user having to explicitely indicate which lemma's where used and which subexpressions were the one affected.

5.3. Interesting language extensions

Now that I've got the *aut* checker, I think there are three things that I can do next.

First of all I can go write a better, more involved Automath checker. This *aut* program is just a one-pass filter and not a 'proof development environment' at all. It's what TeX is for text – a program that goes over the file once – and not a 'word processor' (which is really much easier to write text with.) So a possibility would be to write an *interactive* checker – maybe called 'MacAutomath' – an application with an integrated folding editor that 'knows about Automath'. Furthermore I would then be able to build a checking engine that works incrementally. In the current system, after one changes just one character the only way to find out what this meant for the correctness of the text is to check everything from the start again: which is not very efficient.

Second, I can go *use* the checker. The only significant example it has been run on thus far is the Grundlagen. It would be a nice experience to write a significant piece of Automath[27] myself. This would probably give me a clearer view of where the 'bottlenecks' of the Automath language (and more in general of type theoretical proof checking) really are: I now have lots of ideas about that but those aren't based on any real experience.

Thirdly, I could design my own language, putting in what I currently think are worthwile additions and enhancements. That way, when I start writing a significant piece of text I won't feel cramped by what might be an obsolete technology (although I think that, even after all those years, Automath still is one of the more interesting systems around.)

Now the first option probably would be a lot of fun because programming is fun, designing a user interface is fun and trying to create a checking engine that works incrementally would be fun too. But I don't think it would bring the state of the art of

26. Cf. [O75].

27. Presumably AUT-68: I've never been very fond of AUT-QE.

proof checking much further: it would be research into programming environments really. The second option is really the most interesting: I don't think one should go try innovate things before one has a pretty good idea of what needs change, that is where the main problems are. The third option is what I'll now be presenting a few ideas about.

From what I've seen of Automath thus far there are three aspects of it that seem problematical to me:

1. Modularity

When one looks at the current examples of a large digital proof (not only in Automath: in any system really), it turns out to be a *stream.* It consists of a long sequence of more or less equal sized *small* pieces. This makes it quite hard to get an overview of such a file: it's hard to know where to look in order to get a 'top-down' view of such a document. Now this is not the way someone thinks of a piece of mathematics. Suppose a proof is something like: 'here are two concepts and this (theorem) is how they're related.' Then a digitalization of that proof should – in my view – consists of three 'modules' that say something like: 'this is the first concept (long stream of detail that can be skipped if one knows the concept and believes that the proof checker was right that the detailed description is correct), this is the second concept (another long elaboration that anyone in their right mind would skip) and here is the proof of this theorem (a third elaboration: probably even the main bulk of the text.) So in my opinion, a major feature of a language to do proof checking in would be to have a modular structure that will make it easy to find out what can safely be skipped without losing understanding of the whole.

Now Automath already has a modular structure in the form of its paragraph system, but it's not really suited to this kind of use. This will be apparent from the example I showed of Jutting's work in section 1.2 because there the actual final proof of 'Satz 137'[28] was *not* inside the paragraph '3137', and if it had been, well, something would have to be taken out of it to have the statement of Satz 137 be visible outside that paragraph. So there is a way to 'put something inside a module' in Automath, but it's not very clear how to interface information between the inside and the outside of the module. Furthermore, because there's no explicit way to 'hide' information that's defined inside a paragraph, it won't be clear when reading an Automath text exactly what part of what's inside the 'module' will be used later on.

What one would really like is a modular structure that might be used in the following way: at the start of the module it states what the module is going to deliver ('in this module we define this and this, and then we prove such and such properties of it') and then the only thing which should be relevant is that what's promised is delivered – there even might be several interchangable 'implementations' of the definition and proof content.

2. Naming

While the previous point was about reading a digital proof (which is why I put it first: a text generally has one writer but many readers) this one is about writing it.

28. *trmore(pl(ksi,zeta),pl(eta,zeta),pl(eta,upsilon),t1".3137",t2".3137")*

I think *the* big problem of writing an Automath text is naming things. This is because the concepts and proofs in such a text are built as a long stream of small definitions, each in terms of each other, and every of these items has to have a name in order to be able to refer to it. Jutting solved this by naming each 'step' in his proofs with a sequence number, i.e., he in each proof he had *t1*, *t2*, and so on,[29] but I don't think that's an optimal solution really. He himself seems to have felt this problem as well, as he writes ([NGV94], D.3, page 731):

> *Some of the names I have used lack expressive power. This is partly due to the fact that AUT-QE admits only alphanumeric identifiers, but mainly to my excessive preference for short names.*

Of course, only because he did use such short unexpressive names[30] he was able to finish the job in the first place because of the enormous amount of use that is made of them.

To me it is strange to have to *make up* a name for every property that's refered to. If a statement – for instance – says that 'every value of the function f is positive', then I think that 'the statement that every value of the function f is positive' would be a perfect name for that. Furthermore, when we refer to such a statement in a context where it is clear what statement is needed to complete a proof, I don't think that even stating it explicitely should be necessary: some formal equivalent of *'as we already know'* should be enough.[31]

3. Real math

De Bruijn claims that ([NGV94], A.7, page 215):

> *During the design of Automath I never had the feeling to be implementing a logical system. I just followed the style of the mathematical trade. Since more than a century that style had flourished as a very dependable way of handling mathematical language,* […]

Still, when I look at 'mathematical language' as one encounters it in 'real life', one immediately notices two types of *typographical* structures that I won't be able to find in an Automath text. (Incidentally, both kinds of structure exactly correspond to the way labels are given to certain pieces of the text.)

First, the running text of a mathematical argument is regularly interrupted by a so-called *displayed equation,* which is a centered formula separated from the surrounding text by blank lines, which often has a number in the right margin for identification (in the LaTeX mathematical text processor this corresponds to text between *\begin{equation}* and *\end{equation}* markers).

29. The largest *t* value that occurs is *t243* (in the proof of Satz 283, which says that permuting a sequence of complex numbers doesn't change its sum or product; but that chain really has 247 *t*-notions in it, because it also contains *t33a*, *t54a*, *t87a*, *t87b*, *t88a*, *t88b*, *t89a* and *t121a* while *t84*, *t137*, *t138* and *t162* seem to be missing.)

30. Although he might have decided between English, Dutch and Latin before starting: now we find *not, wel* and *esti.*

31. No, I don't know how to realize this idea either.

Second, a mathematical text very often is structured as a chain of 'items' with names like *definition, axiom, lemma, proposition, theorem, corollary, example.*[32] These are a kind of 'microsection' with each one generally defining just one notion or proving just one statement. They can be recognised by having the name of their kind together with some sequence number in bold face at the start, and they often consist of two parts with the second half being marked *proof* in the same typographical style as the label of the whole, and ending with an end-of-proof marker like a small square. (In LaTeX these kind of 'items' are written using the *\newtheorem* command.)

These two ubiquitous structures in math are immediately apparent in a mathematical text, even when one looks at the page from such a distance that reading the text itself is impossible. Now in Automath there is nothing that corresponds to these two kinds of item. Although the original version of the Grundlagen clearly had both structures in abundance, only the second kind can be found in the translation and even then only in simulation (using the paragraph system.) And despite *something* being present it is not a priori clear from the Automath text where the translation of one 'Satz' ends and where the next one starts, because parts of the translation of such an item are outside the paragraph used to indicate it.

To me the desire to base Automath-like languages on common mathematical practice justifies trying to put these kinds of structural element into those languages.

References

[B91] N.G. de Bruijn, 'Telescopic mappings in typed lambda calculus'. *Information and Computation* 91, 1991, pp. 189-204.

[BM88] Robert Boyer, J. Strother Moore, *A Computational Logic Handbook,* Perspectives in Computing, Vol. 23. Academic Press, ISBN 0121229521, Boston, etc., 1988.

[BW88] H. Boehm, M. Weiser, 'Garbage Collection in an Uncooperative Environment'. *Software Practice & Experience,* September 1988, pp. 807-820.

[DFHHMPPW93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, Benjamin Werner, *The Coq Proof Assistant User's guide,* Version 5.8. Projet Formel, INRIA, Rocquencourt, Lyon,1993.

[DST93] J. H. Davenport, Y. Siret, E. Tournier, *Computer Algebra : Systems and Algorithms for Algebraic Computation.* 2nd Edition, Academic Press, ISBN 0122042328, Boston, etc., 1993.

[F60] Hans Freudenthal, *Lincos,* Design of a Language for Cosmic Intercourse, Studies in Logic and the Foundation of Mathematics. North-Holland Publishing Company, Amsterdam, 1960.

[F93] Scott French, *Just This Once.* Birch Lane Press, Carol Publishing Group, ISBN 1559721731, Secaucus, N.J., 1993.

[J79] L.S. van Benthem Jutting, *Checking Landau's 'Grundlagen' in the Automath System,* Mathematical Centre Tracts 83. Mathematisch Centrum, ISBN 9061961475,

32. Or: *Satz.*

Amsterdam, 1979

[L30] Edmund Landau, *Grundlagen der Analysis.* First Edition 1930, Chelsea Publishing Company, New York, 1965.

[LP92] Zhaohui Luo, Robert Pollack, *LEGO Proof Development System: User's Manual.* Department of Computer Science, University of Edinburgh. Edinburgh, 1992.

[M87] *Macintosh Programmer's Workshop 2.0 Reference.* Apple Computer, Inc., Cupertino, California, 1987.

[M93] William McCune, *Otter 3.0 Reference Manual and Guide,* Draft. Argonne National Laboratory, Argonne, Illinois, 1993.

[NGV94] R.P. Nederpelt, J.H. Geuvers, R.C. de Vrijer (eds.), *Selected Papers on Automath,* Studies in Logic and the Foundations of Mathematics, Vol. 133. Elsevier Science, ISBN 0444898220, Amsterdam, etc., 1994.

[O75] William Orr, 'Euclid Alone'. Copyright 1975 by Damon Knight, In: Rudy Rucker (ed.), *Mathenauts,* Tales of Mathematical Wonder. New English Library, Hodder and Stoughton, ISBN 0450502538, Kent, London, 1989.

[R92] Piotr Rudnicki, *An Overview of the MIZAR Project.* Department of Computing Science, University of Alberta, Alberta, 1992.

[S96] Dan Synek, *CHalf, C implementation of Half for Emacs.* Department of Computing Science, Chalmers University of Technology and University of Göteborg, Göteborg, 1996.

[WR27] Alfred Whitehead, Bertrand Russell, *Principia Mathematica.* First Edition 1910, Second Edition 1927, Cambridge University Press, ISBN 052109187X, London, New York, 1978.