A new implementation of Automath

Freek Wiedijk (freek@cs.kun.nl) *
Department of Computer Science, University of Nijmegen, the Netherlands

Abstract. This paper presents `aut`, a modern Automath checker. It is a straightforward re-implementation of the Zandleven Automath checker from the seventies. It was implemented about five years ago, in the programming language C. It accepts both the AUT-68 and AUT-QE dialects of Automath. This program was written to restore a damaged version of Jutting's translation of Landau's *Grundlagen*. Some notable features:

- It is fast. On a 1GHz machine it will check the full Jutting formalization (736K of non-whitespace Automath source) in 0.6 seconds.
- Its implementation of λ-terms does not use named variables or de Bruijn indices (the two common approaches) but instead uses a graph representation. In this representation variables are represented by pointers to a binder.
- The program can compile an Automath text into one big 'Automath single line' style λ-term. It outputs such a term using de Bruijn indices. (These λ-terms cannot be checked by modern systems like Coq or Agda, because the λ-typed λ-calculi of de Bruijn are different from the Π-typed λ-calculi of modern type theory.)

The source of `aut` is freely available on the Web at the address `<http://www.cs.kun.nl/~freek/aut/>`.

Keywords: Automath, formalized mathematics, proof objects, type theory.

1. Introduction

1.1. Automath

This paper describes an implementation of the mathematical language Automath. The Automath project was the first significant attempt to use a computer to verify the correctness of mathematical texts that have been spelled out in full detail. The Automath languages were designed by N.G. de Bruijn in the late sixties. The Automath project was very active during the seventies, but it died when its funding stopped. However, Automath keeps having a strong influence on the current practice of mathematical proof checking.

The standard reference about Automath is (Nederpelt et al., 1994). This is a compilation of almost all important Automath publications. It contains as (A.3) a good introduction to Automath, (van Daalen,

* Research supported by EU working group 'TYPES' and EU network 'Calculemus'.

© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

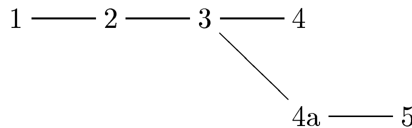
1973). The main important Automath paper that is missing from this collection is the paper (de Bruijn, 1991) about telescopes.

The Automath language has several variants. The main dialects of the language are AUT-68, AUT-QE, AUT-SYNT, AUT-Π, AUT-SL and AUT-ΔΛ. The first two have been implemented. The third and fourth are improvements of the first two that never have been implemented, although texts have been written in them. The last two are variants that are not intended for writing texts, but for compiling texts into.

1.2. Grundlagen

The largest existing Automath formalization is the translation of a small mathematics book. The book is *Grundlagen der Analysis* by Edmund Landau (Landau, 1965). It treats the definition of the field of complex numbers. It consists of 161 pages of German text, which is divided in 301 'Sätze' (theorems). The translation was done by Bert van Benthem Jutting. It is reported on in his PhD thesis (van Benthem Jutting, 1979).

The *Grundlagen* consists of five chapters, which correspond to $\mathbb{N}^+$, $\mathbb{Q}^+$, $\mathbb{R}^+$, $\mathbb{R}$ and $\mathbb{C}$. There are two translations of chapter 4. The first translation follows the German original and defines $\mathbb{R}$ as the union of two disjoint copies of $\mathbb{R}^+$ (for the positive and negative numbers) and of a singleton set (for the zero). The second translation, which is called 4a, defines $\mathbb{R}$ as equivalence classes of pairs of elements of $\mathbb{R}^+$, where two pairs are considered equivalent if they have the same difference. The structure of the Automath translation of the *Grundlagen* is:



This means that the translation of chapter 5 should be checked as a continuation of the second translation of chapter 4.

The file `grundlagen.aut` consists of the concatenation of 1, 2, 3, 4a and 5. It contains 10702 lines of Automath and its size is 736K. A TEX version of the German original is 189K. This means that the Automath version of the book is a factor of 3.9 larger than the original. The Automath file does not contain any whitespace so it makes more sense to compare the sizes of these files when they are compressed with `gzip`. In that case the factor becomes 155K/42K = 3.7. In (Wiedijk, 2000) this factor is called the *de Bruijn factor*. (For the other formalizations that are investigated in (Wiedijk, 2000) the de Bruijn factor also lies around 4.)

1.3. A new implementation

In the early nineties, I got a stack of $5\frac{1}{4}$-inch 'soft' floppy disks containing a copy of the Automath files of the *Grundlagen* through my friend Hans Mulder, who at that time was sharing an office with Bert van Benthem Jutting. These files had been copied from computer to computer during the years, and somehow had gotten corrupted. At several points in the files some bytes would be missing, apparently without any discernible pattern.[1] Together with the floppies I got a copy of (van Benthem Jutting, 1976), which is a five volume technical report containing a full print-out of the non-corrupted files.

The original Automath checkers already were no longer functional at that time. They had been written in a programming language that did not exist anymore, for a computer that did not exist anymore. A simple `yacc` parser quickly pointed out most of the corruption, after which I typed in the missing pieces from the print-out. However, to be sure that I had repaired the files correctly, I needed a new Automath checker.

1.4. Modern systems

(Wiedijk, 2002) is a comparison of fifteen systems for formalization of mathematics in the spirit of the Automath project. These systems are HOL, Mizar, PVS, Coq, Otter, Isabelle, Agda, ACL2, PhoX, IMPS, Metamath, Theorema, Lego, Nuprl and Ωmega. Of these systems Coq, Agda and Lego are closest to Automath. However they are sufficiently different from Automath, that they are not usable for checking the Automath version of the *Grundlagen*. See Sections 6.1 and 6.5 below for a description of the difference between the type system of Automath and that of the modern systems.

Several of these systems – HOL, PVS, Coq, Isabelle, PhoX, Lego and Nuprl – are also descendants of the LCF system (Gordon et al., 1979). They are interactive systems that operate on a 'proof state' by applying 'tactics'. This is very different from Automath, where one writes λ-terms that are checked by a non-interactive checker.

The Agda system from Sweden, implemented by Catarina Coquand, uses a combination of the Automath approach and the LCF approach. Its tactics do not operate on a proof state, but build the λ-term in the text. Both its type theory and its 'user experience' are the closest that one can get to Automath in a serious modern system.

[1] Specifically there were $14 + 15 + 33 + 35 + 32 + 59 + 36 + 35 + 27 + 53 + 64 + 33 + 35 + 21 + 42 = 534$ bytes missing. Because of this, five lines had been joined.

The Coq system from France (The Coq Development Team, 2002), originally implemented by Gérard Huet and Thierry Coquand, and now being developed by a team under direction of Christine Paulin, is the most popular of the Automath descendants. Various pieces of mathematics have been formalized in Coq. For instance there are: a development of real analysis by Micaela Mayero, a development of category theory by Amokrane Saïbi, a proof of the correctness of Buchberger's algorithm by Laurent Théry and Henrik Persson, a constructive proof of the fundamental theorem of algebra by Herman Geuvers, Randy Pollack, Freek Wiedijk and Jan Zwanenburg, and a significant part of the proof of the four color theorem by George Gonthier and Benjamin Werner.

2. The program

We will now describe the `aut` program. We will first present a fragment of Automath to give an impression of the language that the program checks. Then we will describe the implementation of the program and list its features.

2.1. Example of a fragment of Automath

Here is the text of 'Satz 137' (both the statement and the proof) from page 75 of the German version of the *Grundlagen*:

Satz 137: *Aus*

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

folgt

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

Beweis: Nach Satz 134 ist

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

und

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

also

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

The statement of 'Satz 134' that this proof refers to is 'Aus $\xi > \eta$ folgt $\xi + \zeta > \eta + \zeta$.' And here is the Automath translation of 'Satz 137', which is called `satz137`:[2]

[2] The Automath fragment that is shown here corresponds to lines 3330, 3379, 3405, 3553, 3844–3849 of the file `grundlagen.aut`.

```
      * ksi     :=  ---              ; cut
  ksi * eta     :=  ---              ; cut
  eta * zeta    :=  ---              ; cut
 zeta * upsilon :=  ---              ; cut
upsilon * m     :=  ---              ; more(ksi,eta)
    m * n       :=  ---              ; more(zeta,upsilon)

+3137
    n * t1      := satz134(ksi,eta,zeta,m)
                                     ; more(pl(ksi,zeta),
                                       pl(eta,zeta))
    n * 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

    n * satz137 := trmore(pl(ksi,zeta),pl(eta,zeta),
                   pl(eta,upsilon),t1".3137",t2".3137")
                                     ; more(pl(ksi,zeta),
                                       pl(eta,upsilon))
```

From this example it will be clear that an Automath text or *book* consists of *lines*. Each of these lines has four parts: a *context* part, an *identifier* part, a *middle* part and a *category* part. There are three kinds of lines: *block opening* lines (these are the lines that have `---` as the middle part), *primitive notion* lines (these are lines that have `PN` as the middle part, which do not occur in this example) and *abbreviation* lines.

Note that the identifier of a block opening line does not just denote a variable, but also a whole context. A block opening line extends such a context with a new variable. For instance when used in the context part, the identifier `zeta` represents the context `ksi`, `eta`, `zeta` $\vdash \ldots$

The text contains a small *paragraph* called `3137` (this identifier is a combination of chapter 3 and 'Satz' 137). The line `+3137` opens it and the line `-3137` closes it again. The *index* `".3137"` is used to refer to identifiers inside the paragraph.

This Automath fragment defines the three functions `t1(ksi,eta,zeta,upsilon,m,n)`, `t2(ksi,eta,zeta,upsilon,m,n)` and `satz137(ksi,eta,zeta,upsilon,m,n)`. These correspond to the three steps in the proof of 'Satz 137'. We will now briefly explain the meaning of the

function `t1`. The meanings of `t2` and `satz137` are similar. The meanings of the functions occurring in the middle parts of their definitions can be guessed by looking at the German original of this text.

In Automath, mathematical objects have types, and their types have type `TYPE`. Proofs also have types (corresponding to propositions), and these propositional types have type `PROP`.

In this example the variable `ksi` represents a positive real number, and its type `cut` (an abbreviation of 'Dedekind cut') represents the type of positive real numbers. The type of the type `cut` itself is `TYPE`. The function `t1` represents a proof of the statement $\xi + \zeta > \eta + \zeta$. It is a 'proof object' with type `more(pl(ksi,zeta),pl(eta,zeta))`. This type represents a proposition and has itself type `PROP`.

The expression `t1(ksi,eta,zeta,upsilon,m,n)` is an abbreviation of the expression `satz134(ksi,eta,zeta,m)`. Both expressions have the same type: `more(pl(ksi,zeta),pl(eta,zeta))`. This abbreviation means that whenever a well-typed expression `t1(`ξ`,`η`,`ζ`,`υ`,`m`,`n`)` is encountered, it behaves exactly as if it had been the expression `satz134(`ξ`,`η`,`ζ`,`m`)`.

Not all arguments of `t1` are ordinary mathematical objects: the last two arguments are *proof objects*. In contrast with classical mathematics, in Automath these can be used as arguments of functions. The argument `m` is a proof object of the propositional type `more(ksi,eta)` and `n` is a proof object of type `more(zeta,upsilon)`. Therefore `t1` maps proofs of $\xi > \eta$ and $\zeta > \upsilon$ to a proof of $\xi + \zeta > \eta + \zeta$ (the second proof argument is not used in the body of `t1`, and it is not needed for the inference, but it is present in the parameter list). In logical terms this means that `t1` corresponds to the sequent:

$$\xi > \eta, \;\; \zeta > \upsilon \;\vdash\; \xi + \zeta > \eta + \zeta$$

Note that Automath is a rather primitive language. An Automath text just consists of a sequence of abbreviations of λ-terms. There is no proof automation at all. Still the Grundlagen translation manages to faithfully follow the German original.

2.2. Implementation

The `aut` checker is a straight-forward re-implementation of the program that is described in (Zandleven, 1973). Algorithmically it does not contain any new ideas.

`aut` is written in highly portable ANSI C. It is a one pass batch program that reads an Automath book from the standard input stream and prints error messages on the standard error stream. For the lexer and parser of `aut`, the standard tools `flex` and `yacc` have been used.

The source code of the `aut` is 3048 lines long, including the input files for `flex` and `yacc`. Writing the program took about one month of work. It was written about five years ago in the spare time of the author, and took most of a Christmas vacation.

2.3. Features

The `aut` program has the following features:

- `aut` knows both the AUT-68 and AUT-QE languages. See Section 3 below for the details of the languages that `aut` can check.

- To type check, `aut` has to check the convertibility of certain types (the type of a term versus the type that the context of that term expects). To establish this convertibility, `aut` applies β-, δ- and η-reductions. It follows a similar reduction strategy to the one in the Zandleven checker. `aut` can trace reductions with the `-t` flag. It can turn off the η-reductions with the `-e` flag.

- The `aut` program can print various summaries. The flags that are related to this are:

flag	*information printed*
`-d`	duration of the check in seconds
`-l`	counts of the various kinds of lines in the book
`-m`	amount of memory used
`-r`	reduction counts for the various kinds of reduction
`-v`	version of the program

  The `-Z` flag is an abbreviation of the combination `-dlmrv`. With this flag, for the file `grundlagen.aut` the summary that will be printed is:[3]

```
8583 beta reductions, 20004 delta reductions, 2 eta reductions
32 + 6878 = 6910 definitions, 4297 + 6910 = 11207 lines
96 blocks = 3069 kilobytes
0 seconds = 0 minutes 0 seconds
aut 4.2, (c) 1997 by satan software
```

- If an Automath text is not correct, the correctness check will generally take a very long time. This means that Automath correctness, although theoretically decidable, is in practice only semi-decidable.

[3] The '`0 seconds`' means that the check stayed within the same second.

A correct line generally can be checked with only a few reductions[4] but to establish the incorrectness of an incorrect line, `aut` will often need many reductions, and for all practical purposes the check will behave as if the program hangs. (The reason why Automath has this problem and more modern systems do not, is that in Automath definitions are 'transparent' by default. Therefore Automath will keep expanding definitions for a long time when a line is not well typed.)

`aut` can limit the number of reductions per type check with the `-n` flag. If the number of reductions for a type check exceeds this limit, the program will give a message that the line probably is incorrect and then will continue with the rest of the file.

- The `aut` program offers the possibility to have a definition be 'opaque': in that case it will not be used in δ-reductions (transparent and opaque function definitions are also present in the Coq system). Opaque abbreviation lines are indicated by a `~` symbol in front of the middle part of the line. With the `-f` flags these opaque definitions are considered to be transparent again.

- The `aut` program will print the Automath book in a standard form on the output if it is given the `-y` flag. This is not a pretty-printer, as the printed Automath will not contain any white space. With the `-k` flag the user can control whether implicit arguments should be omitted or printed (see Section 5.1 below for a discussion of implicit arguments).

  The original Automath from the seventies could extract an *excerpt* for a line in an Automath book. `aut` will do this if it is given the `-x` flag. See Section 5 below for an example of such an excerpt. `aut` can also generate an excerpt for all primitive notion lines (the 'axioms') in the Automath book.

- `aut` will 'compile' an Automath book to one big λ-term if it is given the `-g` flag. See Section 6 below for a description of this feature.

3. The languages

The `aut` program both knows the AUT-68 and AUT-QE dialects of the Automath language, but it also can check languages that are 'in between' AUT-68 and AUT-QE.

[4] The most complicated line in `grundlagen.aut` is line 9832, which needs 74 reductions.

There are various flags that control various features of the language that is checked as described in Section 3.3 below. With all flags in the 'AUT-68 position' `aut` will check AUT-68, with all flags in the 'AUT-QE position' `aut` will check AUT-QE, but with some flags in one position and some in the other, `aut` will check a language that is in between these two languages.

3.1. Language flags

There are four flags that affect the type theory of the language that `aut` checks.

`-a` *Allow abstractions of degree one.*

This flag has to be chosen for AUT-QE.

Without this flag, there are no λ-abstractions allowed of which the body has degree one, i.e., of which the body is `TYPE` or `PROP`. In `grundlagen.aut` such an abstraction occurs for instance in lines 221–222, which is the definition of the universal quantifier:

```
      * sigma := --- ; TYPE
sigma * p     := --- ; [x,sigma]PROP
    p * all   := p   ; PROP
```

The term `[x,sigma]PROP` in the second line is not allowed without the `-a` flag. (The Automath notation `[x,sigma]` both is used for λ- and Π-abstraction. See Section 6.1 below for a discussion of the identification of λ and Π in Automath.)

`-b` *Omit abstractions when calculating categories of degree one.*

This flag has to be chosen for AUT-68.

Consider the following fragment of the AUT-68 text (D.1) from page 689 of (Nederpelt et al., 1994):

```
{1.1}     * bool  :=   PN              ; TYPE
{1.2}     * x     :=   ---             ; bool
{1.3}   x * TRUE  :=   PN              ; TYPE

{1.4}     * CONTR := [v,bool]TRUE(v) ; TYPE
```

The expression `TRUE(v)` has category `TYPE`. Without the `-b` flag the expression `[v,bool]TRUE(v)` gets the category `[v,bool]TYPE`. With the `-b` flag the abstraction `[v,bool]` is omitted and it gets the correct category `TYPE`.

`-p` *Allow the* `PROP` *type.*

This flag has to be chosen for AUT-QE.

Automath has two basic 'types of types' called `TYPE` and `PROP`. The reason for distinguishing between the two is that without this distinction the 'propositions as types' interpretation causes the double negation law to behave like a Hilbert choice operator.

Without the `-p` flag only `TYPE` is allowed. With this flag, also `PROP` may be used.

`-q` *Allow type inclusion.*

This flag has to be chosen for AUT-QE.

The AUT-QE language has type inclusion. An expression that has category $[x_1, A_1]\ldots[x_n, A_n]$TYPE, also is correct for category $[x_1, A_1]\ldots[x_k, A_k]$TYPE when $k < n$. With the `-q` flag `aut` checks the text with type inclusion.

3.2. Some other flags

There are some other flags that affect the language that `aut` checks. However, these flags are not type related.

`-c` *Put context parameters in front of other definitions.*

Consider the following Automath fragment:

```
    * A := PN  ; TYPE  {1}
    * x := --- ; A     {2}
  x * y := --- ; A     {3}
+p                     {4}
    * x := PN  ; A     {5}
  y * f := x   ; A     {6}
-p                     {7}
```

The `x` on the sixth line, is it the `x` from the second line or is it the `x` from the fifth line? The `aut` program uses the same name space for the identifiers in all three kinds of lines. Therefore the `x` in the definition of `f` is taken to be the `x` from the fifth line, because that is the last definition of `x` that is in scope. However to be able to check the *Grundlagen* translation correctly, the parameters of the definition have to be given priority over the order of the definitions in the file. Therefore with the `-c` flag, `aut` considers the `x` from the second line to be the meaning of the `x` in the definition of `f`.

`-o` *Disallow explicit paragraph references in parameters.*

With this flag, a parameter of a function definition is not allowed to be explicitly qualified with an index that refers to a paragraph.

`-s` *Disallow paragraph re-openers without a star.*

With this flag, `aut` will insist on the `*` in the `+*p` line that re-opens a paragraph `p` that has been closed before.

3.3. AUT-68, AUT-QE, AUT-ΔΛ

The three Automath languages that are relevant for `aut` are AUT-68, AUT-QE and AUT-ΔΛ.

`aut` will accept a language that is in the intersection of AUT-68 and AUT-QE if it is invoked with no flags at all.

`aut` will check the input as AUT-68 with the `-b` flag.

`aut` will check the input as AUT-QE with the `-Q` flag. This is an abbreviation of the combination of flags `-acopqs`. However the *Grundlagen* translation already will be accepted with only the `-acpq` flags.

`aut` cannot check AUT-ΔΛ. However, it can compile its input to an AUT-ΔΛ term, as described in Section 6 below. This term will only make sense as AUT-ΔΛ for the language that `aut` checks with the `-a` flag.

3.4. Syntactical variants

In the Automath literature one finds various syntactical conventions. `aut` tries to be forgiving in this respect, and accepts different notations for the same notions. For instance, it accepts as alternatives:

`*`	↔	`@`	
`:=`	↔	`=`	
`---`	↔	`EB`	
`PN`	↔	`PRIM`	
`;`	↔	`:`	
`;`	↔	<u>`E`</u>	(to be typed as ‘`_^HE`’)
`[x,A]`	↔	`[x:A]`	
`TYPE`	↔	`'type'`	
`.`	↔	`-`	

Furthermore `aut` allows one to switch the order of the middle and category parts. Instead of writing:

```
x * f := t : A
```

one is allowed to write:

```
x * f : A := t
```

This is the natural way to write this line considering its AUT-$\Delta\Lambda$ translation, because in that translation a binder `[f:A]` occurs.

Finally, `aut` considers the context part of a line to be a first class citizen that stands on itself. At each point in the text there will be a 'natural' context. A line:

```
x * f := t : A
```

will consist of two 'halves':

```
[x *]  [f := t : A]
```

that can occur separately on their own. The first half 'sets' the context to `x`. The second half defines the function `f` in the context that is current at that point (note that this line does not have a `*` at the start). If there are several successive abbreviation lines without a context part, they share the same context. After a block opening line the context will be set to the variable that is introduced in that line.

A block opening line (now without context part):

```
x := --- : A
```

also may be written:

```
[x:A]
```

The idea of first class context parts and the alternative syntax for block opening lines both have been taken from the *Grundlagen* files. See Section 5.1 below for an example of an Automath text in this style.

When the context part does not need to be written for all lines, it becomes ambiguous what the contexts are when one changes paragraphs. Consider the following Automath fragment (as not all lines in this fragment have a context part, not all of these lines have a `*`):

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

There are different choices possible for the 'natural' contexts of `f` and `g`. `aut` takes the context after the first `-p` to be the context before the `+p` (instead of the context from 'inside' the paragraph `p`), and the context after the `+*p` to be the context just before this `+*p` (instead of 'remembering' the old context from inside the paragraph `p`). This means that it takes the context of `f` to be `x` and the context of `g` to be `z`. This is consistent with the way that contexts behave in the *Grundlagen* files.

4. Efficiency

We will compare the speed of the `aut` checker with the checkers from the seventies. We will then discuss some implementation issues that affect the efficiency of the `aut` checker.

4.1. Reduction counts and timings

The `grundlagen.aut` file contains 32 primitive notion lines, 4297 block opening lines and 6878 abbreviation lines. Here are some statistics on the number of reductions and the checking times needed for this file, for the two Automath checkers from the seventies and for the `aut` checker:

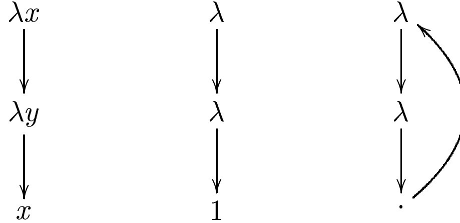
	first checker	second checker	`aut` checker
α-reductions	8952	–	–
β-reductions	5485	6362	8583
δ-reductions	16912	18939	20004
η-reductions	2	2	2
checking time	2112.8 s	4116.1 s	0.6 s

The second checker from the seventies and the `aut` checker both do not use a named representation of bound variables in λ-terms, so they do not need α-reductions. The difference in reduction counts are caused by the difference in reduction strategies of those checkers.

Clearly the computers have become much faster since the seventies! The speed of `aut` is compatible with the speed of the Metamath system (Megill, 1997), which is able to check its whole library in seconds. This system also is a one pass batch checker for a simple proof language and it is also written in C.

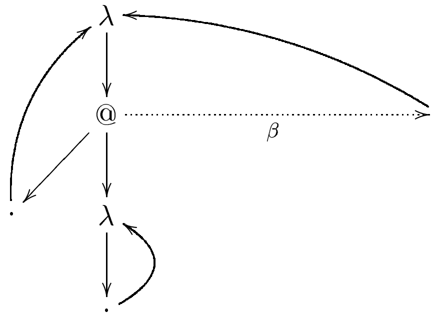
4.2. Term representation

The two common approaches to implement a type checker for typed λ-calculus is using *named variables* or using *de Bruijn indices*. These are the natural choices when using a functional programming language like ML or Haskell. The `aut` checker takes the less common approach (in type checking of typed λ-calculi) of using a *term graph representation*. This is the natural choice when using a procedural programming language like C or Java. Here is the K combinator $\lambda xy.x$ in these three different representations (in this paper we count de Bruijn indices from zero, cf. item 2 in Section 6.2 below, so the middle representation has a 1 instead of a 2):



An advantage of this third representation is that one does not need to be concerned with α-conversions, and also that one does not need to renumber de Bruijn indices when substituting a term inside another term.

An interesting property of the term graph representation is that one will get terms in which variables point to λ-nodes that do not 'see' these variables. I.e., variables will not always be in the sub-term that is 'under' their λ-node. For instance, suppose we have a term $\lambda x.(\lambda y.y)x$, and want to calculate the β-reduct of the sub-term $(\lambda y.y)x$. Now if we do not reduce the full term but just calculate the reduct of the sub-term, then the term graph that will be in memory after the reduction will be:



(The dotted arrow does not correspond to something in memory: it just shows where the β-reduction has happened.) The node for the result

of the reduction, x, will still point to the original λx-node, but it will not be in the sub-term that is under this λ.

`aut` treats its term graphs in a purely functional way: it will only generate new nodes, but it will never modify parts of the graph that already exist.

4.3. Memory management

The `aut` checker is a non-interactive one pass program. This means that its memory management can be extremely simple. It does not need a garbage collector. It also does not need to free memory block-by-block.

The `aut` program allocates memory on a stack. It will never free memory blocks explicitly, so this stack will just grow. When it finishes parsing a line and starts the type check of that line, it remembers the current position of the stack pointer. Then when it finishes type checking the line it resets the stack pointer to this remembered position, effectively freeing all memory that was used during the type check, and moves on to the next Automath line. This means that the memory consumption of the program will grow in a 'saw tooth' pattern. It also means that memory allocation and de-allocation will be fast.

5. Printing excerpts

The `aut` program can extract an *excerpt* of a line in an Automath book. This is the minimal subset of the lines in the book that contains the given line and in which all references from the lines in the subset point to lines in the subset. We will present an example of such an excerpt.

5.1. Excerpt of Satz 1

Here is the text of 'Satz 1' (both the statement and the proof) from page 27 of the German version of the *Grundlagen*:

Satz 1: *Aus*

$$x \neq y$$

folgt

$$x' \neq y'.$$

Beweis: Sonst wäre

$$x' = y'$$

also nach Axiom 4

$$x = y.$$

This 'Axiom 4' that this proof refers to is the fourth Peano axiom. And here is the excerpt of `satz1`[5] from the Automath translation:

```
+l
[a:PROP][b:PROP]
imp:=[x,a]b:PROP
[c:PROP][i:imp(a,b)][j:imp(b,c)]
trimp:=[x,a]<<x>i>j:imp(a,c)
@con:=PRIM:PROP
a@not:=imp(con):PROP
+imp
b@[n:not(b)][i:imp(a,b)]
th3:=trimp(con,i,n):not(a)
-imp
@[sigma:TYPE]
+e
[s:sigma][t:sigma]
is:=PRIM:PROP
+st
+eq
+landau
+n
@nat:=PRIM:TYPE
[x:nat][y:nat]
is:=is(nat,x,y):PROP
nis:=not(is(x,y)):PROP
@suc:=PRIM:[x,nat]nat
ax4:=PRIM:[x,nat][y,nat][u,is(<x>suc,<y>suc)]is(x,y)
[x:nat][y:nat][n:nis(x,y)]
+21
[i:is(<x>suc,<y>suc)]
t1:=<i><y><x>ax4:is(x,y)
-21
satz1:=th3"l-imp"(is(<x>suc,<y>suc),is(x,y),n,[u,is(<x>suc
,<y>suc)]t1"-21"(u)):nis(<x>suc,<y>suc)
-n
-landau
-eq
-st
-e
-l
```

This is a correct AUT-QE text on its own.

[5] The definition of `satz1` is line 980 of the file `grundlagen.aut`.

When printing an Automath text, by default aut omits as many implicit arguments as possible. Implicit arguments are an Automath feature: one is allowed to omit initial arguments to a function if they happen to coincide with the corresponding parameters in the definition of the function. For instance consider the term imp(con) in the definition of the not function. This really is the term imp(a,con) but because imp is defined in the form imp(a,b), the first argument of definition and usage are the same and so the a may be omitted. In this specific example the usage of implicit arguments is only confusing, but implicit arguments become essential in Automath when one works with many function definitions that share a large part of their context.

This excerpt shows that Automath has two kinds of function application. There is function application on the level of the λ-calculus (for instance in the term <<x>i>j, which means '$j(i(x))$'), and there is instantiation of defined functions (for instance in the term imp(a,b)). The first kind is written with angular brackets, it has only one argument, and this argument precedes the function (cf. item 4 in Section 6.2). The second kind is written with round brackets, there can be more than one argument, and these arguments follow the function symbol. In AUT-$\Delta\Lambda$ (as well as in most modern higher order proof assistants) both kinds of function application have been merged.

6. Automath proof objects

The aut checker can *compile* an Automath book to an AUT-$\Delta\Lambda$ proof object. We will describe AUT-$\Delta\Lambda$ and then present an example of such an Automath proof object.

6.1. λ-TYPED VERSUS Π-TYPED TYPE THEORY

Automath is fundamentally different from the modern type theories, which are called *pure type systems* (Barendregt, 1992). In Automath the type of a λ-expression is itself again a λ-expression. In a pure type system the type of a λ-expression is a *product type* or Π-expression.

The Automath type theories traditionally have been described in an algorithmic way (by presenting a type checker), unlike the pure type systems which are generally described by a deduction system of typing rules. In (de Groote, 1993) the Automath type theories are presented in a more modern style. The abstraction rule of a pure type system:

$$\frac{\Gamma, x{:}A \vdash B : C \quad \Gamma \vdash \Pi x{:}A.C : s}{\Gamma \vdash \lambda x{:}A.B : \Pi x{:}A.C}\ s \in \mathcal{S} \qquad (abstraction)$$

then becomes:

$$\frac{\Gamma, x{:}A \vdash B : C}{\Gamma \vdash \lambda x{:}A.B : \lambda x{:}A.C} \qquad (abstraction)$$

This clearly is simpler. The Automath application rule from (de Groote, 1993) can be written as:

$$\frac{\Gamma \vdash F : G \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : Ga}\ G \equiv \lambda x{:}A.B \qquad (application)$$

(This rule suggests to add Π-*application* to pure type systems, with a reduction rule:

$$(\Pi x{:}A.B)a \to B[x := a]$$

Such a system is investigated in (Kamareddine and Nederpelt, 1996) and (Kamareddine et al., 1999). In a sense this system is in between Automath and the pure type systems.)

We will briefly discuss the lack of popularity in the type theoretic community of the λ-typed type theories in Section 7.2 below.

6.2. AUT-ΔΛ

If one streamlines Automath to its simplest form, one ends up with a system that on page 32 of (Nederpelt et al., 1994) is called AUT-ΔΛ. This system is defined in (de Bruijn, 1987), where it is just called ΔΛ. In (de Groote, 1993) the system is called ΛΔ.

In the AUT-ΔΛ system a text is just one big λ-term. For this reason an earlier version of this system was called 'Automath single line' or AUT-SL.

An AUT-ΔΛ term is built from only four primitives:

1. *The type of types.* In (de Bruijn, 1987) this is written as τ. In the proof objects printed by `aut`, it is printed as `*` if it corresponds to `TYPE` and `+` if it corresponds to `PROP`.

2. *Bound variables.* These are represented by de Bruijn indices. Both the paper which introduced the de Bruijn indices (de Bruijn, 1972) and the AUT-ΔΛ paper (de Bruijn, 1987) count them from 1, but the `aut` checker counts them from 0. Also, the `aut` checker represents the number 0 by the empty string. As an example, the term $\lambda A^* a^A.A$ is written $[\tau]\,[1]\,2$ in (de Bruijn, 1987) and is printed `[*][]1` by `aut`.

3. *λ-abstraction.* The term $\lambda x{:}A.B$ is written $[A]\,B$ in (de Bruijn, 1987). It is printed `[`A`]`B by `aut`.

4. *Function application.* The term fa is written $\langle a \rangle f$ in (de Bruijn, 1987). It is printed `<`a`>`f by `aut`, unless it has been derived from the `:=` in an abbreviation line, in which case it is printed `(`a`)`f. This means that to the Automath input line `x := t ; A` corresponds a substring `(t)[A]` in the AUT-$\Delta\Lambda$ term.

To illustrate the smallness of the number four: the basic datatype `exp` of the `aut` checker has seven constructors, the basic datatypes `typ` and `term` of the Isabelle/Pure logical framework together have nine constructors, and the basic datatype `constr` of the Coq system has sixteen constructors.

6.3. Example

The AUT-$\Delta\Lambda$ translation of an `aut` text only makes sense if the text is written in AUT-QE without type inclusion. This is the language that `aut` checks with the `-a` flag.

Here is an example of a text that already is accepted by `aut` without any flags. It is in the intersection of AUT-68 and AUT-QE:

```
+minimal
     * Prop        :=   PN          ; TYPE
     * p           :=   ---         ; Prop
   p * q           :=   ---         ; Prop
   q * imp         :=   PN          ; Prop

+intuitionistic
     * con         :=   PN          ; Prop
   p * not         := imp(con)      ; Prop
-intuitionistic

   p * Proof       :=   PN          ; TYPE
   q * _q          :=   ---         ; [_p,Proof(p)]Proof(q)
  _q * imp_intro   :=   PN          ; Proof(imp)
   q * _p          :=   ---         ; Proof(p)
  _p * _imp        :=   ---         ; Proof(imp)
_imp * imp_elim    :=   PN          ; Proof(q)

+*intuitionistic
+classical
   p * _nn         :=   ---         ; Proof(not(not))
 _nn * notnot_elim :=   PN          ; Proof
-classical
```

```
   p * _con         :=   ---         ; Proof(con)
_con * con_elim     := notnot_elim"-classical"(imp_intro
                         (not,con,[_not,Proof(not)]_con))
                                     ; Proof
-intuitionistic
-minimal
```

And here is the proof object that `aut` prints for this text:

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

Written in conventional λ-notation this is the term:

$$
\begin{array}{l}
\lambda P^{*}\, i^{\lambda p^P q^P . P}\, c^{P} . (\lambda n^{\lambda p^P . P}\, R^{\lambda p^P . *}\, I^{\lambda p^P q^P \hat{q}^{\lambda \hat{p}^{Rp} . Rq} . R(ipq)} \\
E^{\lambda p^P q^P \hat{p}^{Rp} \hat{\imath}^{R(ipq)} . Rq}\, D^{\lambda p^P \hat{n}^{R(n(np))} . Rp} . (\lambda C^{\lambda p^P \hat{c}^{Rc} . Rp} . C) \\
(\lambda p^P \hat{c}^{Rc} . Dp(I(np)c(\lambda \hat{n}^{R(np)} . \hat{c}))))(\lambda p^P . ipc)
\end{array}
$$

Note that in this term the type $P \to P \to P$ of the implication i is $\lambda p^P q^P . P$ instead of $\Pi p^P q^P . P$.

6.4. The proof object of the Grundlagen

The proof object that `aut` prints for an Automath text that uses type inclusion, is not correct as an AUT-$\Delta\Lambda$ term. Although there is no system 'AUT-$\Delta\Lambda$-QE' in the literature, `aut` can still print the proof object. Here is the proof object for `grundlagen.aut`:

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

1.8M *of proof term taking* 32722 *lines omitted*

```
><<3>92>1976>2000]<<1><2>1991><<<><1><2><3><4>6><<1><2>199
6><<3><4>1996>1986><<3><4>1990><1><<<1><2>1996>1994><<<3><
4>1996>1994><3><2965>6711)[[2960][2961][2962][2963][<<<24>
<<>92>1904><<1>92>1976><<<24><<2>92>1904><<3>92>1976>2000]
<1><3>2964]
```

This λ-term contains 2917 τ-types, 96528 λ-abstractions, 406023 function applications, and 499635 bound variables.

6.5. Checking AUT-QE using Coq?

The λ-terms of AUT-QE cannot be checked by a modern system like Coq, Agda or Lego. Consider the following correct AUT-QE text:

```
* nat  :=   PN       ; TYPE
* 0    :=   PN       ; nat
* seq  := [n:nat]nat ; [n:nat]TYPE
* nat0 := <0>seq     ; TYPE
* id   := [n:nat]n   ; seq
```

In this example, the type `seq` represents $\mathbb{N} \to \mathbb{N}$, the infinite sequences of natural numbers. The type `nat0` is the type of the initial element of such a sequence (and it is convertible with `nat`). The function `id` is the sequence corresponding to the identity function: $0, 1, 2, \ldots$

In Automath the λ- and Π-binders have been merged, but in a pure type system they have to be distinguished. It is not possible to choose between a λ- or Π-binder for the binder in `[n:nat]nat` in such a way that the resulting text will be correct in a modern system. It has to be a λ to make the typing `[n:nat]nat` : `[n:nat]TYPE` legal, and also to make the expression `<0>seq` correct, but it has to be a Π to make the typing `[n:nat]n` : `seq` legal.

In the AUT-QE language types are not unique because of the feature of type inclusion. The type of `seq`, `[n:nat]TYPE`, is included in the type `TYPE`. Therefore `seq` can both be used in a context that needs type `[n:nat]TYPE`, as well as in a context that needs type `TYPE`. In a singly sorted (or 'functional') proper type system types are unique (up to convertibility), and there is nothing corresponding to type inclusion.

The problem with the example is not related to type inclusion. It is also present in AUT-$\Delta\Lambda$, a system that does not have type inclusion. The proof object of the example is:

```
[*][]([1]2)[[1]*](<1>)[*]([3])[1]
```

or in conventional λ-notation:

$$\lambda N^* z^N.(\lambda S^{\lambda n^N.*}.(\lambda M^*.(\lambda i^S.i)(\lambda n^N.n))(Sz))(\lambda n^N.N)$$

This is a correct AUT-$\Delta\Lambda$ term. But again, there is no way to change some of the λs in this term to Πs in such a way that it becomes correct in a pure type system. The type of S can not be a sort, so S should not occur as a type, but it does, as the type of i.

In Examples 5.2.4 of (Barendregt, 1992) a singly sorted pure type system called λAUT-QE is defined, taken from (van Benthem Jutting, 1990). It has the following specification:

$\mathcal{S}$ $*, \Box, \Delta$

$\mathcal{A}$ $* : \Box$

$\mathcal{R}$ $(*,*,*), (*,\Box,\Box), (\Box,*,\Delta), (\Box,\Box,\Delta), (*,\Delta,\Delta), (\Box,\Delta,\Delta)$

This system encodes the distinction between the two kinds of function application of AUT-QE, but it does not solve the problem of the disambiguation of λs into λs and Πs, and it does not solve the problem of how to deal with Automath's type inclusion in a pure type system.

It might be possible to generate Coq, Agda or Lego versions of an AUT-QE text (like the Grundlagen translation), but it is not a mere matter of syntax. For this reason, the `aut` program does not have a feature to print its λ-terms in the syntax of one of those systems.

If an approach is developed to generate modern style λ-terms from an AUT-QE text, an implementation of such an approach can start from the proof object that `aut` prints.

7. Conclusion

7.1. Possible future extensions

Here are some possible extensions of the `aut` checker. They are the features that are currently missing. We have no specific plans to implement these in the near future.

- *Telescopes and segments.*

  Automath can be extended with *telescopes*, which are variables for lists of λ-binders. This extension of the Automath language is described in (de Bruijn, 1991). A generalization of telescopes is called *segments* (Balsters, 1986).

  The `aut` checker currently does not implement telescopes or segments. It is not immediately clear how to implement them in the term graph representation that is used by `aut`.

- AUT-SYNT, AUT-Π.

  There are dialects of the Automath language that have never been implemented. In Appendix 9 of (van Benthem Jutting, 1979), reproduced as (B.5) in (Nederpelt et al., 1994), the AUT-SYNT language is described. In this language parts of terms are synthesized automatically by the system. In Chapter VIII of (van Daalen, 1980), reproduced as (B.6) in (Nederpelt et al., 1994), the AUT-Π language is described. This is an extension of AUT-SYNT that has telescopes and Π- and Σ-types.

  The `aut` system is currently not able to check these languages.

7.2. Lessons for modern systems

We do not try to advocate an Automath revival. Automath is very elegant, but it is too basic for the formalization of large scale mathematics. However, our experience with `aut` pointed out some things that might be noted by modern systems:

- *Proof checkers can be fast.*

  `aut` can check the translation of a full book in under a second. The modern systems cannot even come close to that. The reason for this is that the modern systems not only check the proof, but also do a lot of work for the user constructing the proof.

  Obviously they should: the user's time is much more valuable than the computer's time. However, it is desirable to be able to recheck proofs with the speed of `aut` after they have been processed a first time. Most modern systems construct all proofs from scratch every time they recheck a text.

  We advocate investigating the notion of *proof caching.* After a system has constructed a proof it should be kept, and during a next check it then already will be available and can be checked fast.

- AUT-$\Delta\Lambda$ *is an interesting logical framework.*

  The Grundlagen translation uses classical logic, but N.G. de Bruijn claims that Automath is a *restaurant.* Just as in a restaurant one can order different kinds of food, one can use Automath to define different logics. In modern terminology this means that he proposes to use Automath as a *logical framework* (Pfenning, 1996).

  Of the modern systems for formalizing mathematics, the Isabelle system (Paulson, 1994; Nipkow et al., 2002) is the primary one that is based on a logical framework. It consists of the logical framework called Isabelle/Pure, and on top of this a higher order logic is defined called Isabelle/HOL. Also first order logic with ZFC set theory is available as Isabelle/ZF.

  AUT-$\Delta\Lambda$ is the ultimate logical framework. It has not had the attention that it deserves. The two main reasons for this are:

  - The type theory of AUT-$\Delta\Lambda$ is λ-typed, while the current type theories are Π-typed. Because of this, AUT-$\Delta\Lambda$ has no naive set theoretic model and therefore is considered to be 'just syntax' and to have 'no meaning'.

- The theory of LF as a logical framework is well developed, but the corresponding theory of AUT-$\Delta\Lambda$ is not. In (Harper et al., 1991) it has been proved that the proofs that one can do with first order logic encoded in LF correspond exactly to the proofs of first order logic itself. A similar result for AUT-$\Delta\Lambda$ has never been proved.

We claim that it is worthwhile to investigate models of AUT-$\Delta\Lambda$ and to develop the theory for AUT-$\Delta\Lambda$ corresponding to (Harper et al., 1991).

References

Balsters, H.: 1986, 'Lambda calculus extended with segments'. Ph.D. thesis, Eindhoven University of Technology.

Barendregt, H.: 1992, 'Lambda calculi with types'. In: S. Abramsky, D. Gabbay, and T. Maibaum (eds.): *Handbook of Logic in Computer Science*, Vol. II. Oxford University Press, pp. 117–309.

de Bruijn, N.: 1972, 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem'. *Indigationes Math.* **34**, 381–392. (C.2) in (Nederpelt et al., 1994).

de Bruijn, N.: 1987, 'Generalizing Automath by means of a lambda-typed lambda calculus'. In: D. Kueker, E. Lopez-Escobar, and C. Smith (eds.): *Mathematical Logic and Theoretical Computer Science*, Vol. 106 of *Lecture Notes in Pure and Appl. Math.* New York, pp. 71–92, Marcel Dekker. (B.7) in (Nederpelt et al., 1994).

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

de Groote, P.: 1993, 'Defining λ-Typed λ-Calculi by Axiomatizing the Typing Relation'. Technical Report CRIN 93-R-003, Centre de Recherche en Informatique de Nancy.

Gordon, M., R. Milner, and C. Wadsworth: 1979, *Edinburgh LCF: A Mechanised Logic of Computation*, Vol. 78 of *LNCS*. Berlin, Heidelberg, New York: Springer Verlag.

Harper, R., F. Honsell, and G. Plotkin: 1991, 'A Framework for Defining Logics'. Technical Report ECS-LFCS-91-162, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh.

Kamareddine, F., R. Bloo, and R. Nederpelt: 1999, 'On Π-conversion in the λ-cube and the combination with abbreviations'. *Annals of Pure and Applied Logic* **97**, 27–45.

Kamareddine, F. and R. Nederpelt: 1996, 'Canonical typing and Π-conversion in the Barendregt Cube'. *J. Functional Programming* **6**, 245–267.

Landau, E.: 1965, *Grundlagen der Analysis.* New York: Chelsea Publishing Company, fourth edition. First edition 1930.

Megill, N. D.: 1997, 'Metamath, A Computer Language for Pure Mathematics'. `<http://metamath.org/>`.

Nederpelt, R., J. Geuvers, and R. de Vrijer: 1994, *Selected Papers on Automath*, Vol. 133 of *Studies in Logic and the Foundations of Mathematics*. Amsterdam: Elsevier Science.

Nipkow, T., L. Paulson, and M. Wenzel: 2002, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *LNCS*. Springer.

Paulson, L.: 1994, *Isabelle: a generic theorem prover*, Vol. 828 of *LNCS*. New York: Springer-Verlag.

Pfenning, F.: 1996, 'The Practice of Logical Frameworks'. In: H. Kirchner (ed.): *Proceedings of the Colloquium on Trees in Algebra and Programming*. Linköping, Sweden, pp. 119–134, Springer-Verlag LNCS 1059. `<http://www.cs.cmu.edu/~fp/papers/caap96.ps.gz>`.

The Coq Development Team: 2002, 'The Coq Proof Assistant Reference Manual'. `<ftp://ftp.inria.fr/INRIA/coq/current/doc/Reference-Manual-all.ps.gz>`.

van Benthem Jutting, L.: 1976, 'A translation of Landau's "Grundlagen" in AUTOMATH'. Technical report, Eindhoven University of Technology.

van Benthem Jutting, L.: 1979, *Checking Landau's "Grundlagen" in the Automath system*, No. 83 in Mathematical Centre Tracts. Amsterdam: Mathematisch Centrum.

van Benthem Jutting, L.: 1990, 'Typing in Pure Type Systems'. Technical report, Dept. Computer Science, University of Nijmegen, Nijmegen.

van Daalen, D.: 1973, 'A description of Automath and some aspects of its language theory'. In: P. Braffort (ed.): *Proceedings of the Symposium APLASM*, Vol. 1. Orsay. (A.3) in (Nederpelt et al., 1994).

van Daalen, D.: 1980, 'The language theory of Automath'. Ph.D. thesis, Eindhoven University of Technology.

Wiedijk, F.: 2000, 'The De Bruijn Factor'. `<http://www.cs.kun.nl/~freek/notes/factor.ps.gz>`.

Wiedijk, F.: 2002, 'The Fifteen Provers of the World'. `<http://www.cs.kun.nl/~freek/notes/comparison.ps.gz>`.

Zandleven, I.: 1973, 'A verifying program for Automath'. In: P. Braffort (ed.): *Proceedings of the Symposium APLASM*, Vol. I. Orsay. (E.1) in (Nederpelt et al., 1994).