Technische Hogeschool, Eindhoven
Department of Mathematics
Notitie 22, 11 May 1971.

AUT-SL, a single line version of AUTOMATH

by N.G. de Bruijn.

1. Introduction

We can write AUTOMATH in a form as studied by Mr. R. Nederpelt ([2]), called λ-AUTOMATH. This amounts to abolishing block openers and indicator strings, writing everything in the form of abstractions; some of these abstractions are not necessarily ligitimate in AUTOMATH itself. Moreover, expressions like $b(\Sigma_1, \Sigma_2, \Sigma_3)$ are replaced by things like $\{\Sigma_3\}\{\Sigma_2\}\{\Sigma_1\}$ B, where B is not the same as b, but related to B by means of obvious abstractions.

We can go further along this line: First we can abolish all definitions, i.e. all letters expressed by means of a middle other than PN or EB. A next step is to abolish PN's: Being under the reign of a given axiom can be interpreted as living in a block where the axiom is a block opener. For example, the book following

```
bool := PN     type
|b :=  ——     bool                    (1)
|TRUE := PN    type
```

can be compared with a book preceded by the quantifier string

[bool,type][true,[x,bool]type]. (2)

The latter seems to be, in some sense, stronger than (1). The form (2) has the following feature: If we have some model for (1) (i.e. a type β and a mapping τ attaching a category to every object with category β), then the rest of the book can be applied to that model just by substitution. In the form (1), however, such conclusions can only be made metamathematically: every proof of the book following (1) can be rewritten, by trivial translation, as a proof of a statement about such a model.

Since we are abolishing all definitions, or, what is the same thing, abolishing all abbreviations, any interesting line of an AUTOMATH book can now be written and read independently of the preceding book: all necessary information is to be condensed in that line.

There is a further extension, to be explained now. In AUTOMATH we have three kinds of expressions. First, there is a single 1-expression,

viz. the expression "type". (In AUT-QE, an extension of AUTOMATH, we have also 1-expressions like [x,nat][y,bool]type.) Next there are 2-expressions, as Σ in a line like

a := Σ type.

Finally, we have 3 expressions like Θ in

a := Σ type
b := Θ Σ .

But we do not admit, in AUTOMATH, 4-expressions like Γ in

a := Σ type
b := Θ Σ
c := Γ Θ .

In AUT-SL, there is no such restriction. In AUT-SL, we write quantifier strings like

[a,type][b,a][c,b][d,c]

corresponding to

```
| a := —— type
||b := —— a
|||c := —— b
||||d := —— c
```

There are two aspects in which AUT-SL deviates from AUTOMATH. First we do not consider η-conversion in AUT-SL. That is, we do not admit reduction of [x,A]{x}f to f (if x is not free in f). It would not be hard, however, to modify AUT-SL such as to admit η-conversion. A second difference seems to be more serious. In AUTOMATH we have the following. If the book contains

```
|x := —— A
|q := ... type
```

we may write

... := [t,A]q(t) type. (3)

In AUT-QE, however, we may write (3) as well as (4):

... := [t,A]q(t) [t,A]type (4)

In AUT-SL we are more strict: we allow (4) but we forbid (3). This convention makes the language somewhat simpler. It seems to be a good strategy to study this simpler case in every detail before returning to AUTOMATH.

We shall define AUT-SL by means of a program that checks the correctness of an AUT-SL line. The program produces the "normal form" of a given expression, and it produces, if possible, the category of that expression. It either accepts or rejects: it can be shown that the program will never run indefinitely. The proof of this termination property will be given with the aid of a norm, i.e. a positive integer attached, in a particular way, to each accepted expression. The definition of that norm is also given in the program. This norm has been derived by simplification of a norm suggested by Mr. Nederpelt (his norm is not a number, but an expression).

2. The syntax of AUT-SL

The syntax is very simple. We have two sets of symbols: "dummies" and "signs". The set of dummies is infinite; they are different from the signs. The signs are given by

<sign> ::= , | [|] | { | } | type

We can now define "expressions", "quantifier strings" and "expression tails" recursively:

<expression> ::= <quantifier string><expression tail>
<quantifier string> ::= |[<dummy>,<expression>]<quantifier string>
<expression tail> ::= type |<dummy>|{<expression>}<expression>

Our program will attach a normal form to any expression it finds acceptable. These normal forms are "normal expressions"; their syntax is given by

<normal expr> ::= <normal quant str><normal expr tail>
<normal quant str> ::= |[<dummy>,<normal expr>]<normal quant str>
<normal expr tail> ::= type |<dummy>|{<normal expr>}<normal expr tail>

The consequence is that in normal expressions the "}" is never immediately followed by a "[".

The handling of dummies (i.e. the answer to the question of which occurrences of a dummy are bound by which quantifiers) is as explained in [1], with the simplification that parentheses "(" , ")" do not occur.

3. An example

Assume we offer the following expression to the program:

[bool,type][true,[x,bool]type][nonempty,[ksi,type]bool][a,bool]{{a}true}
[ksi,type]{{ksi}nonempty}true

then this is taken in as Z. The action of the program produces as the normal form, to be called Z_1,

[bool,type][true,[x,bool]type][nonempty,[ksi,type]bool][a,bool]
{{{a}true}nonempty}true

Moreover it produces k = 2, i.e. it says that Z and Z_1 are 2-expressions. And the program produces the category of Z, viz. Z_2:

[bool,type][true,[x,bool]type][nonempty,[ksi,type]bool][a,bool]type

As the norm it presents the value m = 7. We devote a few words to its background: Every dummy has a category: viz. in Z the dummy "nonempty" has the category "[ksi,type]bool". Moreover, we have the difference between binding occurrences of a dummy (i.e. occurrences followed by a comma) and bound occurrences (i.e. all other occurrences).

Mr. Nederpelt's norm is obtained roughly as follows: replace the bound occurrences of dummies by their categories. Carry this on until no further bound occurrences of dummies are left. In this way our expression Z gives rise to

[bool,type][true,[x,type]type][nonempty,[ksi,type]type][a,type]
{{type}[x,type]type}[ksi,type]{{type}[ksi,type]type}[x,type]type

Next we start cancelling parts like {A}[x,B]. So {type}[x,type]type reduces to type. Actually all braces { } can be removed this way: the fact that this can be done is a consequence of the acceptability of Z. What remains is

[bool,type][true,[x,type]type][nonempty,[ksi,type]type][a,type]type

and that is essentially Mr. Nederpelt's norm. It is not an acceptable

expression itself.

The simpler norm m we shall work with, is just the number of occurrences of type in Mr. Nederpelt's norm. There are 7 of them, so m = 7.

4. The program

The program is written in ALGOL 60, with some trivial extensions. It uses words like "expression". "quantifier string" in the same way as ordinary ALGOL 60 uses words like integer and real. There is an input statement Z := read by which the given expression is fed into the program, and there are output statements like print (Z_1) which produces the actual expression that was denoted by Z_1, just like print (k) produces the actual number that was denoted by k.

The program uses the following equality of expressions : A ≡ B means that the expressions A and B can be transformed into each other by means of α-conversion, i.e. just by the very unessential process of changing names of dummies (provided that name changing is not done so clumsily that the relation between a bound occurrence and the binding occurrence of a dummy is disturbed).

The program contains the statements

create new dummy s' ; D':= subst(s := s', D)

which have the following meaning. First, s' is a dummy that has not been used before in the expressions occurring in the program. The second statement means that D' gets as its value the expression we obtain from D if we replace every occurrence of s by s'.

In our syntax an empty string was presented by the simple procedure of writing nothing at all. Since this is not always very clear, we shall write ϕ for the empty string.

The program contains clauses like "if X starts with [", which are not ALGOL but cannot be easily misunderstood. And it contains things like "write X = [u,Y]Z". It means: we know at this stage that X has the form [...,...]... ; now give u,Y,Z the (uniquely determined) values such that indeed X = [u,Y]Z.

The reader will notice that the execution of the program causes quite some duplication of work. Having to choose between simpler program and shorter execution, we preferred the former.

```
begin procedure check (Q, X, X1, X2, k, m); value Q, X;
    quantifier string Q; expression X, X1, X2; integer k, m;
  begin if X is a dummy then
        begin dummy y; quantifier string Q1; expression A, A1, A2; integer ka;
            if Q = φ then goto wrong; write Q ≡ Q1[y,A];
            if y ≠ X then check (Q1, X, X1, X2, k, m)
            else begin check (Q1, A, A1, A2, ka, m);
                       X1:=X; X2:=A1; k:=ka+1
                 end
        end
      else if X ≡ type then begin k:=1; X1:= type; m:=1 end
      else if X starts with [ then
          begin dummy u; expression Y, Y1, Y2, Z, Z1, Z2; integer ky, my, mz;
              write X = [u,Y]Z;
              check (Q, Y, Y1, Y2, ky, my);
              check (Q[u,Y], Z, Z1, Z2, k, mz); m:=my+mz;
              X1:=[u,Y1]Z1; if k > 1 then X2:=[u,Y1]Z2
          end
      else if X starts with { then
          begin expression Y, Y1, Y2, Z, Z1, Z2, W1, W2;
              integer ky, kz, kw, my, mz, mw;
              write X ≡ {Y}Z;
              check (Q, Y, Y1, Y2, ky, my); if ky=1 then goto wrong;
              check (Q, Z, Z1, Z2, kz, mz); if Z1 ≡ type then goto wrong;
              if kz>1 then check (Q, {Y1}Z2, W1, W2, kw, mw);
              if Z1 starts with [ then
                 begin dummy u; expression V, R;
                     write Z1≡[u,V]R; if V≠Y2 then goto wrong;
                     if (R=type)∨(R is a dummy ≠ u)
                        then check (Q, R, X1, X2, k, m)
                     else if R≡u then begin X1:=Y1; X2:=Y2;
                                            k:=ky; m:=my
                                      end
                     else if R starts with { then
                          begin expression C, D; write R≡{C}D;
                              check(Q,{{Y1}[u,V]C}{Y1}[u,V]D,X1,X2,k,m)
                          end
```

```
                    else if R starts with [ then
                        begin dummy s; expression C, D, D';
                            write R≡[s,C]D;
                            create new dummy s';
                            D':=subst(s:=s',D);
                            check(Q,[s',{Y1}[u,V]C]{Y1}[u,V]D',X1,X2,k,m)
                        end
                    else goto wrong;
                end
            else begin X1:={Y1}Z1;
                    if kz=1 then goto wrong; X2 := W2;
                        k:=kz; m:=mw
                end
        end
    else goto wrong
end procedure check;

program:
    expression E, E1, E2; integer k, m; E := read;
      check (φ, E, E1, E2, k, m);
      print(E1); print(k); print(m); if k > 1 then print(E2);
      goto end;
wrong:      print(‡ the given expression is not acceptable ‡);
end:
end
```

5. Termination of the program

The program either accepts of rejects: it does not run forever. We shall sketch a proof for this statement.

Let us look at the most crucial part of the program. It is about $\{Y_1\}Z_1$ where $Z_1 = [u,V]\{C\}D$. Since Z_1 is in normal form, we have $\{C\}D = \{C_n\}\{C_{n-1}\}\ldots\{C_1\}v$ (with $C = C_n$). Since $C_1 \ldots, C_n$ are all "simpler" than Z_1, we may assume that

$$\{Y_1\}[u,V]C_i$$

can be reduced to normal form C_i^*, for $i = 1,\ldots, n$. Now if $v \neq u$, the normal form of $\{Y_1\}Z_1$ is obtained. If $v = u$, we get

$$\{C_n^*\} \ldots \{C_1^*\}Y_1,$$

which is not necessarily normal. Assume

$$Y_1 = [x_1,A_1] \ldots [x_k,A_k]\{E_1\} \ldots \{E_m\}w.$$

The danger arises that w is equal to one of $x_1,\ldots,x_\ell$, where $\ell = \min(k,n)$. This means that at a further stage in the program the w will be replaced by one of the C_i^*, so that we still have no normal form as yet. In order to cope with this difficulty, we remark that the norm of this C_i^* is less than the norm of Y_1. The argument is that this norm of C_i^* equals the norm of A_i (the norm of A_i is unaffected by the substitutions that have taken place in the meantime: $x_1 := C_1^*$, $x_i := C_i^*,\ldots$). And the norm of A_i is obviously less than the norm of Y_1.

The feature we just described, makes it possible to carry out induction with respect to the norm of Y_1: we assume that the program terminates for all $\{Y_1\}Z_1$ if the norm of Y_1 is $< N$ (Y_1 and Z_1 in normal form); next we take a Y_1 with norm N and we assume that the program terminates with $\{Y_1\}Z_1$ for all Z_1 with less than M symbols. Having assumed this, we can deal with the case with Y_1 of norm N and Z_1 with M symbols in the way we described above.

This arrangement of the proof is partly due to Mr. L.S. van Benthem Jutting, working on ideas proposed by Mr. L.E. Fleischhacker.

6. Final remarks

The language AUT-SL has been defined by means of reduction to normal form. The next step is that we can produce a number of language rules that do not involve the normal form explicitly. First, let us call two acceptable expressions definitionally equivalent (symbol: $\overset{D}{=}$) if they have the same normal form. Next we can formulate rules producing definitional equivalence, like:

if $A_1 \overset{D}{=} A_2$, $B_1 \overset{D}{=} B_2$, if $\{A_1\}B_1$ is acceptable,

then $\{A_2\}B_2$ is acceptable, and $\{A_1\}B_1 \overset{D}{=} \{A_2\}B_2$.

In this way, we can phrase quite a number of derived rules for AUT-SL. On the other hand, we could take these derived rules as the definition of AUT-SL (in the same line as the definition of AUTOMATH), and prove the reducibility to normal forms as a theorem.

References

[1] N.G. de Bruijn, On the use of bound variables in AUTOMATH. Technological University, Eindhoven, Internal Report, Notitie 9 (26 November 1970).

[2] R.P. Nederpelt, LAMBDA-AUTOMATH, Technological University, Eindhoven, Internal Report, Notitie 17 (8 January 1971).