Department of Mathematics
Technological University
Eindhoven,
The Netherlands.

REPRINT. Published in the
Proceedings of the Symposium
on APL (Paris, December 1973),
ed. P. Braffort.

A description of AUTOMATH and some aspects of its language theory

by

D.T. van Daalen *)

0. Summary

This note presents a self-contained introduction into AUTOMATH, a formal definition and an overview of the language theory. Thus it can serve as an introduction to the papers of L.S. Jutting [7] and I. Zandleven [11] in this volume. Among the various AUTOMATH languages this paper concentrates on the original version AUT-68 (because of its relative simplicity) and one extension AUT-QE (in which most texts have been written thus far).
The contents are:

1. Introductory remarks.
2. Informal description of AUT-68.
3. Mathematics in AUTOMATH: propositions and types.
4. Extension of AUT-68 to AUT-QE.
5. A formal definition of AUT-QE.
6. Some remarks on language theory.

For a description of the AUTOMATH project and for its motivation we refer to Prof. de Bruijn's paper also in this volume [4].

*) The author is employed in the AUTOMATH project and is supported by the Netherlands Organization for the Advancement of Pure Science (Z.W.O.).

1. Introductory remarks

1.1. According to the claims for the *formal system* AUTOMATH one should be able to formalize many mathematical fields in it in such a precise and complete fashion that machine verification becomes possible. The flexibility required to meet the indicated universality is provided by having a rather meagre *basic system*. The AUTOMATH user himself has to add appropriate *primitive notions* to the basic system in order to introduce the concepts and axioms specific to the part of mathematics he likes to consider. In this respect, the basic system may be compared with some usual system of logic (e.g. first order predicate calculus) to which one adds mathematical axioms in order to form mathematical theories.

1.2. In spite of this analogy however the basic system itself does not contain any logic in the usual sense. Basic for the system are the concepts of *type* and *function* (instead of, e.g., the concept of set or of natural number), which are formalized by a certain *typed λ-calculus*.
When representing mathematics in AUTOMATH one has to deal with the question of *coding*: How to formalize general mathematical concepts in the form of *types* and *functions* (see section 2.2). Clearly an appropriate formalization will incorporate as much as possible of the basic type-and-function framework. Section 3 discusses this coding problem and in particular proposes a suitable way of representing propositions, predicates and proofs (a *functional interpretation* of logic).

1.3. In order to satisfy the claim of automatic verification of correctness the system certainly has to be decidable (and even *feasibly decidable* on now-existing computing machines). Since many common mathematical theories produce undecidable sets of theorems we must conclude that we cannot expect the computer to do all our work. Indeed theorems have to be given *together with their proofs* in order to allow verification.
Thus the correctness produced by the machine verification covers the arguments leading from axioms to conclusions only. The AUTOMATH user himself is responsible for his choice of primitive notions and all the coding (and decoding) involved.

2. Informal description of AUTOMATH

2.1. Introduction

Here we treat the original version of AUTOMATH, now named AUT-68. We chose this system as an example because of its relative simplicity. The discussion will be informal and intuitive and in fact **restricted** to the object-and-type fragment of the language (thus leaving the proof-and-proposition fragment to section 3).

2.2. Intuitive framework

(This section may be skipped by formalists).

The mathematical entities discussed in the language fall into two sorts: *objects* and *types*. The types may be considered as classes or sets of a certain kind, which may have objects as their elements. All types are supposed to be disjoint, for each object belongs to just one type. This *uniqueness of types* permits one to speak about *the* type of an object.

The typestructure is built up by starting from *ground types* and forming *function types* from these. Each mathematician may choose the ground types himself (as primitive notions), e.g. the type of natural numbers.

An example of a function type is the type $\alpha \to \beta$ (where α and β are types) of the functions from α to β. More generally, the function types are formed by taking *products*, as follows: The language allows one to express dependence of types on objects (of some given type). That is, one can describe certain families of types β_x indexed by the objects x of a given type α. Now every function type is formed as the *generalized Cartesian product* of such β_x, usually denoted $\prod_{x\in\alpha}.\beta_x$, and containing as objects just these functions that associate to any object x of type α an object of type β_x. The type $\alpha \to \beta$ is the special case where all β_x are a fixed type β.

2.3. Expressions, degrees and formulas; correctness

The language as such only expresses the constructions of types and objects and the typing relations between objects and types.

The *expressions* of the language have *degree* 1, 2 or 3. Types and objects are denoted by expressions of degree 2 and 3 respectively (for short 2-expressions, 3-expressions). For convenience we introduce the 1-expression type to provide a type for the types. Further 1-expressions will be introduced in sections 3 and 4.

The symbol E expresses the typing relation: ... has type ... . So if A denotes an object then we have the E-formulas A E α and α E type. The 2-expressions and 3-expressions are built up from *variables* and *constant-expressions* by means of:

i) the **substitution mechanism (section 2.5)**

ii) functional abstraction and application (sections 2.8 and 2.10).

The constant-expressions have the form $c(x_1,\ldots,x_k)$ where $x_1,\ldots,x_k$ are variables and c is either a *primitive* constant introduced as a primitive notion (sections 2.6) or a *defined* constant (section 2.7).
Expressions and formulas are *correct* if they are constructed according to the rules of the language, which are informally discussed in the sequel.

2.4. Variables and contexts

A mathematical statement generally presupposes certain assumptions on the variables used. For example: "let x be a natural and y a real number". In AUTOMATH, in accordance with this usage, each variable of degree 3 (*object-variable*) ranges over a certain type, called the type of the variable. The 2-variables (*type-variables*) are supposed to range through the types and have type as their type.
Expressions and formulas containing *free* object- or type-variables, say $x_1,\ldots,x_k$, can only be *correct* relative to a certain *context*: I.e. a finite sequence of E-formulas x_1 E $\alpha_1,\ldots,x_k$ E α_k, called *assumptions*, in which the free variables have to be explicitly introduced with their types.
Some of the types α_i may depend on the variables given earlier in the sequence. For instance, α_3 may contain both x_1 and x_2 as free variables. It is understood that all α_i are correct expressions themselves: α_1 relative to the *empty* context, α_2 relative to x_1 E α_1, etc.

2.5. Substitution mechanism

Let us, in informal discussion, exhibit the possible dependence of an expression Σ on variables $x_1,\ldots,x_k$ by writing $\Sigma[\![x_1,\ldots,x_k]\!]$ *for* Σ. Then we write $\Sigma[\![A_1,\ldots,A_k]\!]$ for the result of *simultaneously substituting* A_i for x_i (for i = 1,...,k) in Σ.

Suppose that under assumptions $x_1 \underline{E} \alpha_1, \ldots, x_k \underline{E} \alpha_k$ we have a correct $\underline{E}$-formula $A[\![x_1,\ldots,x_k]\!] \underline{E} \alpha[\![x_1,\ldots,x_k]\!]$. Then the *substitution mechanism* yields the substitution *instance* $A[\![A_1,\ldots,A_k]\!] \underline{E} \alpha[\![A_1,\ldots,A_k]\!]$ for any sequence $A_1,\ldots,A_k$ of suitable candidates for $x_1,\ldots,x_k$. I.e. these $A_1,\ldots,A_k$ have to be of the appropriate types where, however, in view of the possible dependence of types on variables, the substitution has to take place in the types too. So we require

$$A_1 \underline{E} \alpha_1,\ A_2 \underline{E} \alpha_2[\![A_1]\!],\ldots,A_k \underline{E} \alpha_k[\![A_1,\ldots,A_{k-1}]\!] .$$

2.6. Primitive notions

As mentioned before, one has to add primitive notions to the basic system in order to introduce the specific concepts of the piece of mathematics one wants to study.

For example, in order to write about the natural numbers, one might introduce the primitive *type-constant nat* and the *object-constant* 1 by axiomatically stating:

$$\begin{aligned} &\text{nat } \underline{E} \text{ type} \\ &1 \ \underline{E} \text{ nat} . \end{aligned}$$

In general, primitive notions are introduced by stating an axiomatic $\underline{E}$-formula $p(x_1,\ldots,x_k) \underline{E} \alpha[\![x_1,\ldots,x_k]\!]$ under certain assumptions $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$. Here either α is $\underline{\text{type}}$ (and p is a type-constant) or in the current context we have $\alpha \underline{E} \underline{\text{type}}$ already (p being an object-constant).

All correct substitution instances $p(A_1,\ldots,A_k)$ of such a constant-expression $p(x_1,\ldots,x_k)$ are then produced by the substitution mechanism, described above.

For example, the concept of *successor* in the natural number system can be introduced under the assumption $x \underline{E}$ nat by stating: successor(x) $\underline{E}$ nat.

Using the substitution mechanism we get

$$\begin{aligned} &\text{successor}(1) &&\underline{E} \text{ nat} \\ &\text{successor}(\text{successor}(1)) &&\underline{E} \text{ nat, etc} . \end{aligned}$$

Notice that primitive constant-expressions may not only contain object-variables (like the x in successor(x)) but also type-variables.

2.7. Abbreviations

In mathematics one often introduces abbreviations, i.e. new names for possibly long and complicated expressions. In AUTOMATH this abbreviation facility is also present; indeed, it will appear that by the particular format of the language every *derived* statement gives rise to the introduction of a new defined constant. Although this kind of explicit definition is often considered theoretically uninteresting, we feel that it is essential in practice for the actual formalization and verification of complicated theories.
Just like primitive notions, abbreviations are introduced under certain assumptions and so may contain free variables in general. Thus new constant-expressions $d(x_1,\ldots,x_k)$ are introduced, abbreviating expressions D which are correct in the current context. Clearly the type of $d(x_1,\ldots,x_k)$ must be the same as that of D.
Example: 2,3,... can be introduced by

2 := successor(1)
3 := successor(2), etc .

Further, the notion of "successor of successor" might be abbreviated by stating (under assumption x $\underline{E}$ nat) that

plustwo(x) := successor(successor(x)) .

Again, all correct substitution instances with their types are produced by the substitution mechanism.

2.8. Functional abstraction: λ-calculus

We have mentioned functional abstraction and application as further tools for constructing expressions. By these devices a form of typed λ-calculus is incorporated into the basic system. In λ-calculus, intuitively speaking, λx.B denotes the function which to any object x associates the object B. Or (exhibiting the dependence on x) λx.B[x] is the map which, with any A, associates B[A].
In AUTOMATH (where all functions have a *domain*) such explicitly given functions are denoted by *abstraction expressions* [x,α]B, where B may contain x as a free variable; α is the type of x and the domain of the function. In case B is a 3-expression, [x,α]B attaches objects to the objects of type α and is called an *object-valued function*. If B is a 2-expression, [x,α]B

attaches types to the objects of type α and is called a *type-valued function*. In AUT-68 no abstraction expressions of degree 1 are formed (in contrast with AUT-QE).

Notice that possible free *occurrences* of x in B are *bound* by the abstractor $[x,\alpha]$ and are not free in $[x,\alpha]B$ any more. An important restriction on abstracting is that such a bound variable must be a 3-variable. Thus we only *quantify* (cf. section 3.4) over (the objects of) a given type and quantification over type is not possible.

2.9. Type of abstraction expressions

Suppose that under the assumption $x \underline{E} \alpha$ we have $B \underline{E} \beta$. If β is not a 1-expression then we may form both the abstraction expressions $[x,\alpha]B$ and $[x,\alpha]\beta$. According to section 2.8 $[x,\alpha]B$ denotes an object-valued function and $[x,\alpha]\beta$ denotes a type-valued function.

The latter abstraction expression $[x,\alpha]\beta[\![x]\!]$ however is also used with a different meaning in Automath, that is, to denote the *corresponding function type* $\underset{x\underline{E}\alpha}{\Pi}.\beta[\![x]\!]$ (which is the type of $[x,\alpha]B[\![x]\!]$ by section 2.2).

So we obtain $[x,\alpha]B \underline{E} [x,\alpha]\beta$ and $[x,\alpha]\beta \underline{E}$ type.

Example: the successor *function* can be introduced (in the empty context) by

$$\text{succfun} := [x,\text{nat}]\text{successor}(x) \underline{E} [x,\text{nat}]\text{nat} .$$

The double use of 2-expressions mentioned above does not cause ambiguity, because it is always clear whether an expression acts as a function or as a type in a formula. In fact in AUT-68 abstraction expressions of degree 2 are exclusively used with the second meaning, i.e. as function types.

2.10. Functional application

In full (i.e. type-free) λ-calculus any expression - as a function - may be applied to any expression - even itself - as an argument.

In AUTOMATH, as a *typed* λ-calculus, all functions have *domains* and any form of self-application is ruled out by the *application restrictions*: The *application expression* <A>B (denoting the result of applying B as a function to A as an argument) is correct only if:

i) B is a function and so has a domain, say α.

ii) A is an object of type α.

The notation <A>B, with the argument in front, is somewhat unusual; it is convenient however since abstractions are written in front too.

2.11. Type of application expressions

Assume that B $\underline{E}$ [x,α]β. Here [x,α]β[x] is a 2-expression acting as a type and so denotes $\Pi_{x\underline{E}\alpha}.\beta[x]$. Hence B must be considered as a function with domain α. Now if A $\underline{E}$ α we are allowed to form the application expression <A>B having β[A] as its type.

Note that B need not be of the form [x,α]C itself. It may, e.g., be a single object variable or object constant with type [x,α]β.

Example: As an alternative expression for the number 3 we might introduce

3alt := <2>succfun $\underline{E}$ nat .

2.12. Equality

We will define a relation of *definitional equality* among the correct expressions, appropriate to the interpretation of expressions suggested above. The relation is denoted ... = ... and generated by:

i) *abbreviational* or δ-*equality*, $=_\delta$

ii) λ-*equality*.

The latter is generated in turn by β-*equality*, $=_\beta$, and η-equality, $=_\eta$. Usually in λ-calculus the λ-equality also explicitly embodies α-*equality* (renaming of bound variables). In this note however we take the point of view of simply ignoring the names of the bound variables. So α-equal expressions are identified and are a fortiori definitionally equal by the reflexivity of the = -relation (cf. also section 5.3.2).

2.12.1. δ-equality

Assume the defined constant d has been introduced in suitable context by

$$d(x_1,\ldots,x_k) := D[x_1,\ldots,x_k] .$$

Then $d(x_1,\ldots,x_k)$ abbreviates D and we write $d(x_1,\ldots,x_k) =_\delta D$. And further for the substitution instances:

$$d(A_1,\ldots,A_k) =_\delta D[A_1,\ldots,A_k] .$$

2.12.2. β-equality

Assume <A>[x,α]B[[x]] is a correct expression (so A $\underline{E}$ α). Now β-equality exploits the interpretation of [x,α]B as a function with domain α and simply amounts to evaluating the result of the application:

$$\langle A\rangle[x,\alpha]B =_{\beta} B[[A]] .$$

2.12.3. η-equality

In mathematics one usually considers functions as *extensional* objects, in the sense that functions with the same domain and which are pointwise equal are identified. In AUTOMATH this extensional equality is *partly* covered by the η-equality: *If* x *does not occur free in* B *then* [x,α]<x>B $=_{\eta}$ B (for correct expressions only). This is intuitively sound only if domain B = α, which indeed is the case by the correctness of [x,α]<x>B.

2.12.4. Definitional equality

Now definitional equality = is defined to be the *equivalence relation* on the correct expressions, generated by $=_{\delta}$, $=_{\beta}$, $=_{\eta}$ and by *monotonicity*: *If* A = A' *and* B' *is produced from* B *by replacing one specific occurrence of* A *in* B *by* (*an occurrence of*) A' *then* B = B'.

Or, using *suggestive dots* for the *unchanged* part of the expression B: *If* A = A' *then* ... A ... = ... A' ... .

Example of the monotonicity rule: If A = A' then <C><A>D = <C><A'>D (if both expressions are correct).

2.13. The format: books and lines

2.13.1. Actual AUTOMATH texts are written in the form of books. A *book* consists of a finite sequence of *lines*. Each line must be placed in a certain *context* (the context of the line) and introduces a new identifier of a certain type. All lines consist of four consecutive parts, separated by suitable marks or spaces:

i) *context part*, indicating the context of the line. In general the context part consists of the *context indicator*, i.e. the last variable of the current context. From this the complete context can easily be recovered. If the context of the line is $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$, the sequence of variables $x_1,\ldots,x_k$ is called the *indicator string* of the line. The *empty* context can be indicated by an empty context part.

ii) *identifier part*, consisting of the new *identifier*.

iii) *middle part*, containing the symbol EB (cf. 2.13.2), the symbol PN (cf. 2.13.3) or the *definition* of the new identifier (cf. 2.13.4).

iv) *category part*, containing the type of the new identifier.

Assume an AUTOMATH book is given, in which the variable x_k has been introduced with type α_k in the context $x_1 \underline{E} \alpha_1,\ldots,x_{k-1} \underline{E} \alpha_{k-1}$. Then we may add lines with context indicator x_k, so having $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$ as their context. Below we discuss the three different kinds of lines.

2.13.2. The *block opening lines* have middle part EB (for *empty block opener*) or, in alternative notation, a bar ———. An EB-line introduces a new *variable* and thus allows extension of the current context by one assumption.
Example: $x_k * y := \underline{EB}\ \underline{E}\ \alpha$ ("let y be of type α") introduces a new variable y of type α. Lines having y as their context part - which may appear later in the book - then have $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$, $y \underline{E} \alpha$ as their context.

2.13.3. The *primitive notion lines* have middle part PN and introduce the primitive notions. For example:

$$x_k * p := \underline{PN}\ \underline{E}\ \alpha$$

introduces the primitive constant expression $p(x_1,\ldots,x_k)$ and contains the *axiomatic* E-statement $p(x_1,\ldots,x_k) \underline{E} \alpha$.

2.13.4. The *abbreviation lines* look like:

$$x_k * d := D\ \underline{E}\ \alpha\ ,$$

where the middle part D is the definition of d, i.e. the expression to be abbreviated. This line contains, relative to the preceding book and the current context, both the derived E-statement $D \underline{E} \alpha$ and the defining axiom for the new defined constant d:

$$d(x_1,\ldots,x_k) := D\ .$$

2.14. Correctness of lines; validity

A line is *correct* if both the middle part (if not EB or PN) and the category part are correct expressions with respect to the preceding book and the current context, and the category part is the type of the middle part (if not EB or PN). For the correctness of the expressions, all identifiers used have to be *valid*. Constants are valid in a book from the line on in which they are introduced. Free variables are valid in a line if they occur in its context. We speak about the *block* of lines in which a free variable is valid (whence *block opener*).

2.15. Shorthand facility

Assume that a primitive or defined constant c was introduced in a certain context $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$. Then if later in the book c occurs with fewer than k arguments, the argument list is completed by adding a suitable initial segment of the original indicator string (cf. 2.13.1ii)) $x_1,\ldots,x_k$. In other words the expression $c(A_{i+1},\ldots,A_k)$ is shorthand for $c(x_1,\ldots,x_i,A_{i+1},\ldots,A_k)$ and the single constant c is shorthand for $c(x_1,\ldots,x_k)$. Clearly the completing variables have to be valid, that is, the initial segments of the original and the current context have to coincide. The shorthand facility accords with usual mathematical practice where free variables are often considered as fixed throughout an argument and are not mentioned explicitly.

2.16. Paragraph system

For each variable and constant it must be possible to retrace from which line it originates. This condition is clearly satisfied when all names are unique. A more liberal method of naming however is allowed by the socalled *paragraph system*, for a description of which we refer to Zandleven [11, section 11]. Both shorthand facility and paragraph system do not really concern the language definition but are present for convenience only.

2.17. Example

In the following AUT-68 booklet the examples of the preceding sections are now written in the proper format.

```
    * nat         := PN                      type
    * 1           := PN                      nat
    * x           := ——                      nat
x   * successor   := PN                      nat
    * 2           := successor(1)            nat
    * 3           := successor(2)            nat
x   * plustwo     := successor(successor)    nat
    * succfun     := [x,nat]successor(x)     [x,nat]nat
    * 3alt        := <2>succfun              nat
```

Here the middle part of plustwo uses the shorthand facility. It is left to the reader to establish 3 = 3alt.

3. Mathematics in AUTOMATH: Propositions as types

3.1. Functional interpretation of logic

Up till now we have described AUTOMATH as a calculus of objects and their types only. A major part of mathematics however consists of making statements and reasoning with them, i.e. deals with logic.
Now there are different ways of coding some logic into the objects-and-types framework. Here we only mention a socalled *functional interpretation* of logic, which gives rise to the *propositions-as-types* notion. This idea of interpreting logic was developed independently by de Bruijn and certain others, of whom we mention Howard [6], Prawitz [10], Girard [5] and Martin-Löf [8].

3.2. Propositions as types

So far we have introduced type as the only 1-expression. We had Σ E type and Γ E Σ for the types Σ and the objects Γ of type Σ respectively. Now we introduce another 1-expression, the basic symbol prop. Originally in AUT-68 no distinction was made between type and prop. The latter 1-expression acts just like type and was introduced later to allow difference of treatment between types which are to be considered as propositions and types which are just types of objects.
If Σ E prop we consider Σ as a proposition. If further Γ E Σ, we consider Γ as some construction establishing the truth of Σ (a "proof" of Σ). Thus the formula Γ E Σ is conceived as *asserting* the proposition Σ.

3.3. Interpreting implication

Let α E prop and β E prop. Now we may say we have a "proof" of the implication $\alpha \to \beta$ if from an assumption of the truth of α we can argue and conclude the truth of β. That is, if for any construction establishing the truth of α we can produce a construction for the truth of β or, equivalently, if we have a map from "proofs" of α to "proofs" of β.
Now in AUTOMATH terminology: we say we "prove" $\alpha \to \beta$ if for any x E α we can produce some B E β. I.e. if we have some Σ in the function type $[x,\alpha]\beta$. So we let $[x,\alpha]\beta$ denote the implication $\alpha \to \beta$ and have $[x,\alpha]\beta$ E prop. This corresponds to the second interpretation of abstraction expressions in section 2.9.

Now by this interpretation we obtain the *modus ponens* (from α and $\alpha \to \beta$ infer β) by simple functional *application*. For let $A \underline{E} \alpha$ and $\Sigma \underline{E} [x,\alpha]\beta$ (A and Σ thus being "proofs" of α and $\alpha \to \beta$ respectively). Then by the application rule we construct $\langle A \rangle \Sigma$ establishing the truth of β.

3.4. Universal quantification; negation

In exactly the same manner a function interpretation of *universal* statements can be given. Namely if $\alpha \underline{E}$ **type** and for $x \underline{E} \alpha$ we have $\beta \underline{E}$ **prop** then we identify the function type $[x,\alpha]\beta$ with the universal statement $\forall_{x \underline{E} \alpha} \beta$. Here functional application corresponds to the "*instantiation*" *rule* in logic.
Thus by this interpretation of logic in AUTOMATH one gets the $(\forall,\to)$-fragment of first order predicate logic for free. However in AUTOMATH only positive statements are made and statements like: "Σ is not of type Γ" cannot be expressed. In order to interpret negation we introduce as a primitive notion the proposition *con* (for "contradiction") together with some suitable axiom (primitive notion). Here are different possibilities, e.g. the intuitionistic *absurdity rule* (for any proposition α, from con infer α) or the classical *double negation law*. Then an AUTOMATH theory (i.e. book) is *consistent* if, in the empty context, it does not produce some $\Sigma \underline{E}$ con.
For $\alpha \underline{E}$ **prop** we define *non*(α) as $\alpha \to$ *con* or, in AUTOMATH notation, $[x,\alpha]$*con*.
Now the double negation law can be stated by introducing the primitive notion *dnl* as follows: *If* $\alpha \underline{E}$ **prop**, $x \underline{E}$ *non*(*non*(α)) *then* *dnl*(α,x) $\underline{E}$ α.
By also choosing suitable definitions for the other connectives $(\wedge,\vee)$ and the existential quantifier we can smoothly obtain full classical first order predicate calculus.

3.5. Assumptions, axioms, theorems

In AUTOMATH-books the $\underline{E}$-formula $\Gamma \underline{E} \Sigma$ for proposition Σ can occur in the usual three kinds of lines again:

i) $\underline{EB}$-lines: $\sigma * x := \underline{EB} \underline{E} \Sigma$.
These must be interpreted as *assumptions*: "let Σ hold" or "let x be a proof of Σ". Now in a line where x is valid we may refer to x whenever we want to use the assumed truth of Σ.

ii) $\underline{PN}$-lines: $\sigma * p := \underline{PN} \underline{E} \Sigma$.
These serve as axioms, or rather as axiom *schemes* (by the dependence on the variables contained in the context σ).

iii) abbreviation lines: $\sigma * d := \Gamma \underline{E} \Sigma$ must be considered as derived statements, i.e. theorems, lemmas etc. Here the middle part Γ "proves" the proposition Σ from the assumptions in the context σ.

3.6. Book-equality

The definitional equality (cf. section 2.12) of AUTOMATH only covers a small part of the usual mathematical equality. Further a statement of definitional equality cannot be handled as an actual proposition; e.g. it cannot be negated or even assumed (as in: let A = B). As the AUTOMATH-counter part of the usual mathematical ... equals ..., the *book-equality* IS(α,A,B) - where A and B are objects of type α - can be introduced by suitable primitive notions, some of which are shown in the example below.

$* \alpha$	:= ———	type
$\alpha * x$	:= ———	α
$x * y$	:= ———	α
$y *$ IS	:= PN	prop
$x *$ REFL	:= PN	IS(x,x)
$y * i$	:= ———	IS(x,y)
$i *$ SYM	:= PN	IS(y,x)
	etc.	

and also:

$\alpha * \beta$	:= ———	type
$\beta * f$	:= ———	$[x,\alpha]\beta$
$f * x$	:= ———	α
$x * y$	:= ———	α
$y * i$	:= ———	IS(x,y)
$i *$ ISAX1	:= PN	IS(β,<x>f,<y>f)

By the axiom of reflexivity (REFL) above, definitional equality implies book-equality: if $A \underline{E} \alpha$, $B \underline{E} \alpha$, $A = B$ then REFL(α,A) $\underline{E}$ IS(α,A,B).

4. Extension of AUT-68 to AUT-QE

4.1. Function-like expressions

Expressions Σ such that Σ E $[x,\alpha]\beta$ or $\Sigma = [x,\alpha]\beta$ are called *function-like* expressions. Whereas in AUT-68 function-like 3-expressions may have any form, e.g. they can be variables or primitive constant expressions, the only function-like 2-expressions are (possibly abbreviated) abstraction expressions. This is because function-like 1-expressions are absent in AUT-68.
Thus we can discuss explicitly constructed families of types β_x where x ranges over some type α (namely by forming the abstraction expression $[x,\alpha]\beta[\![x]\!]$) but we cannot discuss *arbitrary* families of types indexed by x E α. Indeed, we cannot introduce a family of types as a primitive notion or as a variable.

4.2. Supertypes or quasi-expressions

In AUT-QE on the other hand such arbitrary type-valued functions are admitted however, by extending the class of 1-expressions. The new 1-expressions, *quasi-expressions* (whence AUT-*QE*) or *supertypes*, have the form $[x_1,\alpha_1] \ldots [x_k,\alpha_k]$ type or $[x_1,\alpha_1] \ldots [x_k,\alpha_k]$ prop, where $\alpha_1,\ldots,\alpha_k$ are 2-expressions, i.e. propositions or types.
For example, an arbitrary type-valued function on α can be introduced by an EB-line:

$$\sigma * f := \text{——} [x,\alpha]\text{type} .$$

If for α we take the type of natural numbers, then f is an arbitrary *sequence of types*.

4.3. The use of AUT-QE

Similarly we have arbitrary *prop-valued functions* in AUT-QE. These are especially useful in our interpretation of logic, for a prop-valued function with domain α is nothing but a *predicate* over α. For example, by an EB-line

$$\sigma * R = \text{——} [x,\text{nat}][y,\text{nat}]\text{prop}$$

an arbitrary binary predicate (rather: relation) on the natural numbers is introduced. The presence of predicate and relation variables in AUT-QE allows us to write *axiom schemes* with such variables, e.g. to introduce a further equality axiom (cf. section 3.6) we can write:

α * P := ——— [x,α]prop

P * x := ——— α

x * y := ——— α

y * i := ——— IS(x,y)

i * j := ——— <x>P

j * ISAX2 := PN <y>P

We emphasize however that abstraction over such 2-variables (e.g. type-variables, prop-variables, predicate-variables) in AUT-QE is still forbidden, so both AUT-68 and AUT-QE may still be called *first-order systems*.

4.4. Type-inclusion and prop-inclusion

Just as in AUT-68 the function-like 2-expression f (cf. section 4.2) also codes its corresponding function space, i.e. the type of those g with domain α such that for A E α we have <A>g E <A>f. As prop behaves just like type, the predicate P (cf. section 4.3) also denotes the proposition $\forall_{x \underline{E} \alpha}.P(x)$.

As a consequence, we allow the transition from Σ E [x,α]type to Σ E type. This transition or, in general, from

$$\Sigma \ \underline{E} \ [x_1,\alpha_1] \ \ldots \ [x_k,\alpha_k][y_1,\beta_1] \ \ldots \ [y_m,\beta_m]\underline{type}$$

to

$$\Sigma \ \underline{E} \ [x_1,\alpha_1] \ \ldots \ [x_k,\alpha_k]\underline{type}$$

is called *type-inclusion*. The similar transition with prop instead of type is called *prop-inclusion*. By this *type-inclusion* and *prop-inclusion* AUT-QE contains AUT-68 as a proper *subsystem*. Notice that for 2-expressions uniqueness of types - if A E α, A E β then $\alpha = \beta$ - is lost.

4.5. Let us finish with a table in which some AUTOMATH notions are listed with their possible meanings in the propositions-as-types interpretation.

AUTOMATH-notions	object-and-type interpretation	proof-and-proposition interpretation
2-expressions	types	propositions
3-expressions	objects	proofs
... E ...	... has type ...	... proves ...
function-like 2-expressions	type-valued functions	predicates
	function types	implications
		universal statements
EB-lines	variable introductions	assumptions
PN-lines	primitive object introductions	axioms
abbreviation lines	definitions or abbreviations	theorems

5. A formal definition of AUT-QE

5.1. The language, to be defined formally now, is the one accepted by the current checker (cf. [11]) except for two points:

i) Paragraph facilities are not present here so all constant names have to be distinct (cf. section 2.16).

ii) There is no shorthand facility (i.e. all expressions are written out in full (cf. section 2.15).

The actual formalism has been chosen in this way in order to keep as close as possible to the preceding informal book-and-line description. A definition along more usual *natural deduction* lines may possibly be more elegant. For technical reasons we preferred to avoid redundancy almost completely in our definition. As a consequence of this, some useful extra rules follow as *derived rules* in the section on language theory.

5.2. Our aim is to define formally what correct AUT-QE books are.
The description consists of:

i) Preliminaries, mainly devoted to the context free part of the language (section 5.4).

ii) *Simultaneous* definition of correctness of books, contexts, lines, expressions, E-formulas and = - formulas (section 5.5).

The = - formulas only serve as a help in our definition; they do not appear in the book. The kernel of ii) is the definition of correctness of expressions and formulas relative to a certain book and context. Here the book serves to determine the set of primitive notions and abbreviations, and the context serves to determine the set of valid free variables.
Most concepts are introduced by *ordinary inductive definitions*. These consist of a finite set of rules of the form: "if ... then ...". Here only such conclusions may be drawn which follow from a finite number of applications of the rules.

5.3. Notational conventions

5.3.1. An extensive use is made of *syntactic variables* throughout the definition. Often certain assumptions on these variables are implicit by their specific choice, e.g. σ and ξ always run over contexts. Syntactic variables may always be indexed or primed.

5.3.2. As for substitution and α-*conversion* (renaming of bound variables) we adopt the following point of view: expressions with bound variables are considered as named versions - named to facilitate reading - of some actually *namefree* skeleton (cf. [3]). Thus we identify α-equal expressions and assume that α-conversion is applied whenever necessary to avoid *clash of variables*. We use ... ≡ ... to denote *syntactic identity* (symbol-for-symbol equality) modulo α-equality. E.g. [x,Σ] ...x...x... ≡ [y,Σ] ...y...y... .

5.3.3. Correctness of expressions A and formulas φ relative to a book $\mathcal{B}$ and a context σ are abbreviated by $\mathcal{B}$; σ ⊢ A and $\mathcal{B}$; σ ⊢ φ respectively. Sometimes we write ⊢ A or σ ⊢ A for $\mathcal{B}$; σ ⊢ A and ⊢ φ or σ ⊢ φ for $\mathcal{B}$; σ ⊢ φ when there is no particular need to emphasize the current book or context. The notions $\vdash^{(i)}$A and $\vdash^{(i)}$A E B are used to express that A is an i-expression and ⊢ A (respectively ⊢ A E B).

5.4. Preliminaries

5.4.1. Alphabet

1) As *variables* and *constants* we allow any *alphanumeric string*. Such a string is considered atomic and is thus counted as one single symbol. Syntactic variables for variables are x,y,z,... Among the constants (syntactic variable c) we distinguish *primitive* (syntactic variables p,q) and *defined* or *abbreviational* constants (syntactic variable d).
2) Improper symbols
   i) Some *brackets* and *braces*: [,], (,), < , >.
   ii) Some *separation marks*: !, *, ⊢, E, :=, =, *semicolon* and *comma*.
   iii) Some *reserved symbols*: EB, PN.

5.4.2. Expressions (syntactic variables A,B,C,D,...,Σ,Δ,Γ,...)

i) *Variables*: x
ii) *Abstraction expressions*: [x,Σ]Δ
iii) *Applications expressions*: <Σ>Δ
iv) *Constant-expression instances*: $c(\Sigma_1,\ldots,\Sigma_k)$
v) *Basic constants*: type, prop.

As special syntactic variables for 2-*expressions* we take α,β,... .

5.4.3. Formulas (syntactic variable φ)

i) $\underline{E}$-*formulas*: $\Sigma\ \underline{E}\ \Delta$

ii) $=$-*formulas*: $\Sigma = \Delta$.

5.4.4. Additional concepts

1) *Contexts* (syntactic variables σ,ξ): *Any finite (possibly empty) sequence of* $\underline{E}$-*formulas* $x_i\ \underline{E}\ \Sigma_i$, *separated by commas, where all* x_i *are different.*

2) Lines (syntactic variable λ)

i) $\underline{EB}$-*lines* : $\sigma * x := \underline{EB}\ \underline{E}\ \Sigma$

ii) $\underline{PN}$-*lines* : $\sigma * p := \underline{PN}\ \underline{E}\ \Sigma$

iii) *Abbreviation lines*: $\sigma * d := \Delta\ \underline{E}\ \Sigma$

3) *Books* (syntactic variable $\mathcal{B}$): *Any finite (possibly empty) sequence of lines, separated from one another by exclamation signs* (!).

5.4.5. Free variables

We define the *free variable* set $FV(\Sigma)$ of expressions Σ by induction on the structure of Σ (cf. section 5.4.2):

i) $FV(x) = \{x\}$

ii) $FV([x,\Gamma]\Delta) = FV(\Gamma) \cup (FV(\Delta)\backslash\{x\})$

iii) $FV(<\Gamma>\Delta) = FV(\Gamma) \cup FV(\Delta)$

iv) $FV(c(\Sigma_1,\ldots,\Sigma_k)) = \bigcup_{i=1,\ldots,k} FV(\Sigma_i)$

v) $FV(\underline{prop}) = FV(\underline{type}) = \emptyset$.

5.4.6. Substitution

1) The result of *simultaneous substitution* of $A_1,\ldots,A_k$ for the free variables $x_1,\ldots,x_k$ in an expression Σ is denoted by $[\![x_1,\ldots,x_k/A_1,\ldots,A_k]\!]\Sigma$ and locally abbreviated by Σ^*:

i) $x_i^* \equiv A_i$

ii) $y^* \equiv y$ if y not among $x_1,\ldots,x_k$

iii) $([y,\Sigma_1]\Sigma_2)^* = [y,\Sigma_1^*]\Sigma_2^*$ if y not among $x_1,\ldots,x_k$ and $(x_i \in FV(\Sigma_2) \Rightarrow y \notin FV(A_i))$ for $i = 1,\ldots,k$ (otherwise rename y in $[y,\Sigma_1]\Sigma_2$).

iv) $(\langle\Sigma_1\rangle\Sigma_2)^* \equiv \langle\Sigma_1^*\rangle\Sigma_2^*$

v) $(c(\Sigma_1,\ldots,\Sigma_k))^* \equiv c(\Sigma_1^*,\ldots,\Sigma_k^*)$

vi) $\underline{\text{prop}}^* \equiv \underline{\text{prop}}$, $\underline{\text{type}}^* \equiv \underline{\text{type}}$.

2) *Substitution* of A for x is denoted by $[\![x/A]\!]$ and amounts to the case k = 1 above.

5.5. Correctness

5.5.1. Correct books

i) *the empty book is correct*

ii) *if* $\mathcal{B}$ *is correct and* λ *is correct with respect to* $\mathcal{B}$ *then* $\mathcal{B}!\lambda$ *correct.*

5.5.2. Correct context with respect to $\mathcal{B}$:

i) *the empty context is correct*

ii) *if* $\sigma * x := EB\ \underline{E}\ \Delta$ *is a line in the book* $\mathcal{B}$ *then* $\sigma, x\ \underline{E}\ \Delta$ *is a correct context with respect to* $\mathcal{B}$.

5.5.3. Correct lines with respect to $\mathcal{B}$:

1) $\underline{\text{EB}}$-lines: *If* $\mathcal{B}$; $\sigma \vdash^{(1)} \Delta$ *or* $\mathcal{B}$; $\sigma \vdash^{(2)} \Delta$, $\sigma \equiv x_1\ \underline{E}\ \Sigma_1,\ldots,x_k\ \underline{E}\ \Sigma_k$, *and* y *not among* $x_1,\ldots,x_k$ *then* $\sigma * y := \underline{\text{EB}}\ \underline{E}\ \Delta$ *is a correct line with respect to* $\mathcal{B}$.

2) $\underline{\text{PN}}$-lines: *If* $\mathcal{B}$; $\sigma \vdash^{(1)} \Delta$ *or* $\mathcal{B}$; $\sigma \vdash^{(2)} \Delta$ *and* p *does not occur in* $\mathcal{B}$ *then* $\sigma * p := \underline{\text{PN}}\ \underline{E}\ \Delta$ *is a correct line with respect to* $\mathcal{B}$.

3) Abbreviation lines: *If* $\mathcal{B}$; $\sigma \vdash \Sigma\ \underline{E}\ \Delta$ *and* d *does not occur in* $\mathcal{B}$ *then* $\sigma * d := \Sigma\ \underline{E}\ \Delta$ *is a correct line with respect to* $\mathcal{B}$.

5.5.4. Correct E-formulas relative to $\mathcal{B}$ and σ

1) Repetition rule: *If* $\sigma \equiv x_1\ \underline{E}\ \Sigma_1,\ldots,x_k\ \underline{E}\ \Sigma_k$ *and* Σ_j *is an* i*-expression then* $\mathcal{B}$; $\sigma \vdash^{(i+1)} x_j\ \underline{E}\ \Sigma_j$ (for j = 1,...,k).

2) Abstraction rule: *If* $\mathcal{B}^* \equiv \mathcal{B}!\sigma * x := \underline{\text{EB}}\ \underline{E}\ \alpha$ *and* $\mathcal{B}^*$ *is correct and* $\mathcal{B}^*$; $\sigma, x\ \underline{E}\ \alpha \vdash^{(i)} \Sigma\ \underline{E}\ \Delta$ *then* $\mathcal{B}$; $\sigma \vdash^{(i)} [x,\alpha]\Sigma\ \underline{E}\ [x,\alpha]\Delta$.

3) Application rules:

i) *If* $\vdash A\ \underline{E}\ \alpha$ *and* $\vdash^{(i)} B\ \underline{E}\ [x,\alpha]C$ *then* $\vdash^{(i)} \langle A\rangle B\ \underline{E}\ [\![x/A]\!]C$.

ii) *If* $\vdash A\ \underline{E}\ \alpha$, $\vdash^{(i)} B\ \underline{E}\ C$ *and* $\vdash C\ \underline{E}\ [x,\alpha]D$ *then* $\vdash^{(i)} \langle A\rangle B\ \underline{E}\ \langle A\rangle C$ (clearly i will be 3 here).

4) Substitution rule: *If* Σ *is an* i*-expression and either*
$x_1 \underline{E} \Sigma_1,\ldots,x_k \underline{E} \Sigma_k * c := \underline{PN} \underline{E} \Sigma$ *or* $x_1 \underline{E} \Sigma_1,\ldots,x_k \underline{E} \Sigma_k * c := \Delta \underline{E} \Sigma$
is a line in the book $\mathcal{B}$ *and* $\mathcal{B}$; $\sigma \vdash A_j \underline{E} [\![x_1,\ldots,x_k/A_1,\ldots,A_k]\!]\Sigma_j$ *for*
$j = 1,\ldots,k$ *then* $\mathcal{B}$; $\sigma \vdash^{(i+1)} c(A_1,\ldots,A_k) \underline{E} [\![x_1,\ldots,x_k/A_1,\ldots,A_k]\!]\Sigma$.

5) Rule of type-conversion: *If* $\vdash \Delta \underline{E} \Sigma$ *and* $\vdash \Sigma = \Gamma$ *then* $\vdash \Delta \underline{E} \Gamma$.

6) Rules of type- and prop-inclusion:

i) *If* $\vdash \Sigma \underline{E} [x_1,\alpha_1] \ldots [x_k,\alpha_k][y,\beta]\underline{type}$ (*possibly* $k = 0$) *then*
$\vdash \Sigma \underline{E} [x_1,\alpha_1] \ldots [x_k,\alpha_k]\underline{type}$.

ii) *If* $\vdash \Sigma \underline{E} [x_1,\alpha_1] \ldots [x_k,\alpha_k][y,\beta]\underline{prop}$ (*possibly* $k = 0$) *then*
$\vdash \Sigma \underline{E} [x_1,\alpha_1] \ldots [x_k,\alpha_k]\underline{prop}$.

5.5.5. Correct expressions with respect to $\mathcal{B}$ and σ

1) Correct 1-expressions:

i) *If* $\mathcal{B}$ *is correct and* σ *is correct with respect to* $\mathcal{B}$ *then* $\mathcal{B}$; $\sigma \vdash^{(1)} \underline{type}$
and $\mathcal{B}$; $\sigma \vdash^{(1)} \underline{prop}$.

ii) *If* $\mathcal{B}^* \equiv \mathcal{B}!\sigma * x := \underline{EB} \underline{E} \alpha$ *and* $\mathcal{B}^*$; $\sigma,x \underline{E} \alpha \vdash^{(1)} \Delta$ *then* $\mathcal{B}$;$\sigma \vdash^{(1)} [x,\alpha]\Delta$.

2) Correct 2- and 3-expressions: *If* $\vdash^{(i)} \Sigma \underline{E} \Delta$ *then* $\vdash^{(i)} \Sigma$.

Remark: It is intended that $\mathcal{B}$; $\sigma \vdash A$ or $\mathcal{B}$; $\sigma \vdash \varphi$ only if $\mathcal{B}$ is correct and σ is correct with respect to $\mathcal{B}$. This condition is explicitly imposed in 5.5.5.1i) and propagated all through the definition.

5.5.6. Correct =-formulas with respect to $\mathcal{B}$ and σ

1) β-equality: *If* $\vdash \langle A\rangle[x,\alpha]B$ *and* $\vdash [\![x/A]\!]B$ *then* $\vdash \langle A\rangle[x,\alpha]B = [\![x/A]\!]B$.

2) η-equality: *If* $\vdash [x,B]\langle x\rangle C$, *and* $x \notin FV(C)$ *and* $\vdash C$ *then* $\vdash [x,B]\langle x\rangle C = C$.

3) δ-equality: *If* $x_1 \underline{E} \Sigma_1,\ldots,x_k \underline{E} \Sigma_k * d := \Delta \underline{E} \Sigma$ *is a line in* $\mathcal{B}$, *and*
$\mathcal{B}$; $\sigma \vdash A_j \underline{E} [\![x_1,\ldots,x_k/A_1,\ldots,A_k]\!]\Sigma_j$ *for* $j = 1,\ldots,k$, *and*
$\mathcal{B}$; $\sigma \vdash [\![x_1,\ldots,x_k/A_1,\ldots,A_k]\!]\Delta$ *then* $\mathcal{B}$; $\sigma \vdash d(A_1,\ldots,A_k) = [\![x_1,\ldots,x_k/A_1,\ldots,A_k]\!]\Delta$

4) Monotonicity rules:

i) *If* $\mathcal{B}^* \equiv \mathcal{B}!\sigma * x := \underline{EB} \underline{E} \alpha$ *and* $\mathcal{B}^*$; $\sigma,x \underline{E} \alpha \vdash B_1 = B_2$ *then*
$\mathcal{B}$; $\sigma \vdash [x,\alpha]B_1 = [x,\alpha]B_2$.

ii) *If* $\vdash \alpha_1 = \alpha_2$, $\vdash [x,\alpha_1]B$, *and* $\vdash [x,\alpha_2]B$ *then* $\vdash [x,\alpha_1]B = [x,\alpha_2]B$.

iii) *If* $\vdash A_1 = B_1$, $\vdash A_2 = B_2$, $\vdash \langle A_1\rangle A_2$, *and* $\vdash \langle B_1\rangle B_2$ *then* $\vdash \langle A_1\rangle A_2 = \langle B_1\rangle B_2$.

iv) *If* $\vdash A_j = B_j$ (for $j = 1,\ldots,k$), *and* $\vdash c(A_1,\ldots,A_k)$, *and* $\vdash c(B_1,\ldots,B_k)$
then $\vdash c(A_1,\ldots,A_k) = c(B_1,\ldots,B_k)$.

5) Reflexivity, symmetry and transitivity rules

i) *If* $\vdash A$, $\vdash B$ *and* $A \equiv B$ *then* $\vdash A = B$

ii) *If* $\vdash A = B$ *then* $\vdash B = A$

iii) *If* $\vdash A = B$, *and* $\vdash B = C$ *then* $\vdash A = C$.

Remark: It is intended that $\mathcal{B}; \sigma \vdash A = B$ only if both $\mathcal{B}; \sigma \vdash A$ and $\mathcal{B}; \sigma \vdash B$. In most cases above, though sometimes unnecessary, such conditions have been explicitly stated. Where they have been omitted it will be immediate that they hold by some other conditions.

6. Some remarks on language theory

6.1. Decidability

The language theory is mainly concerned with the investigation of the basic system. A major aim is to prove the *decidability* of the AUTOMATH languages. That is, to prove the existence of an effective procedure which for any given text in a finite amount of time decides whether it is correct or not (in AUT-QE, say). The kernel of such a *checker* deals with the *verification* of correctness of expressions and formulas (both $\underline{E}$- and =-formulas), relative to a given book and context (which are assumed to be correct already). In this section we shall sketch a certain checking procedure, closely related to the actually running verifying program of Zandleven (cf. [11]). We shall also roughly indicate the proof of correspondence between the proposed checking procedure and the language definition of the preceding section.

6.2. Reduction

6.2.1. In order to study the =-relation in more detail we introduce the *reduction relation* $\geq$, a partial order among the expressions. For an explanation of the suggestive dots in our definition we refer to section 2.12.4.

6.2.2. Definition:

1) *One-step* reduction (with respect to a book B)

i) one-step β-reduction: $\ldots\langle A\rangle[x,\alpha]C\ldots \gt_\beta \ldots[\![x/A]\!]C\ldots$

ii) one-step η-reduction: *If* $x \notin FV(C)$ *then* $\ldots[x,\alpha]\langle x\rangle C\ldots \gt \ldots C\ldots$

iii) one-step δ-reduction: *If* d *was introduced by an abbreviation line* $x_1 \underline{E} \Sigma_1,\ldots,x_k \underline{E} \Sigma_k * d := D \underline{E} \Sigma$ *in* B *then* $\ldots d(\Sigma_1,\ldots,\Sigma_k)\ldots \gt_\delta \ldots[\![x_1,\ldots,x_k/\Sigma_1,\ldots,\Sigma_k]\!]D\ldots$

iv) also $\gt$ is allowed with any *combination* of the indices such as: *If* $A \gt_\beta B$ *or* $A \gt_\eta B$ *then* $A \gt_{\beta\eta} B$

v) one-step reduction *in general*: *If* $A \gt_{\beta\eta\delta} B$ *then* $A \gt B$.

2) *Many-step* reduction (with respect to β)

i) *If* $A \equiv B$ *then* $A \geq B$

ii) *If* $A \geq B$ *and* $B > C$ (*with respect to* β) *then* $A \geq C$.

So $\geq$ is the reflexive and transitive closure of $>$. Likewise $\geq_{\beta\delta}$ denotes the reflexive and transitive closure of $>_{\beta\delta}$ etc. For $A \geq B$ we also write $B \leq A$.

3) i) Reduction sequence: A sequence $\Sigma_1, \Sigma_2, \ldots$ of expressions is called a *reduction sequence* of Σ_1 if for all i we have $\Sigma_i \equiv \Sigma_{i+1}$ or $\Sigma_i > \Sigma_{i+1}$.

ii) Proper reduction sequence: A *reduction sequence* $\Sigma_1, \Sigma_2, \ldots$ is called *proper* if for all i we have $\Sigma_i > \Sigma_{i+1}$.

6.2.3. Clearly the =- relation is the equivalence relation generated by the restriction of $>$ to correct expressions. So we can conclude: $\vdash A = B$ *iff* $A \equiv C_1 \geq D_1 \leq C_2 \geq D_2 \leq \ldots \geq D_{k-1} \leq C_k \equiv B$ (*possibly* $k = 1$), *where all expressions in the respective reduction sequences are correct.*

6.2.4. As an example of a reduction sequence consider:
3alt $>_\delta$ <2>succfun $>_\delta$ <2>[x,nat]successor(x) $>_\beta$ successor(2) $>_\delta$ successor(successor(1)) (see section 2.16). So each reduction step seems to bring us closer to some possible "outcome". Here β- and δ-reduction amount to evaluation and η-reduction to a certain simplification of expressions.

6.3. The three problems: normalization, Church-Rosser and closure

6.3.1. It will appear that the decision procedure for equations (=- formulas) plays a central role in the checker. As first we state - in terms of the remark in section 6.2.4 - two important questions around reduction and definitional equality:

i) (*Normalization*) Do correct expressions always have a final outcome, i.e. do they always reduce to an expression which does not reduce further?

ii) (*Church-Rosser property*) Do definitionally equal expressions have a common outcome, i.e. an expression to which they both reduce?

A third central question concerns the so-called *closure property* (this term was introduced by R.P. Nederpelt in the introduction to [9]):

iii) Is the system closed under reductions, i.e. do correct expressions remain correct under reduction?

6.3.2. Normalization and strong normalization

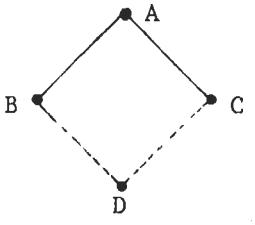
Let us define:

1) A is *normal* if no one-step reduction A > B can be applied.
2) A is said to *normalize* if A reduces to some normal B (which is then called a *normal form* of A).
3) A is said to *strongly normalize* if all proper reduction sequences of A terminate.

We say that *normalization* (resp. *strong normalization*) holds if all correct expressions normalize (resp. strongly normalize). Normalization (and a fortiori strongnormalization) does not hold in the full λ-calculus (take as a counter example the expression <λx.<x>x>λx.<x>x). In typed systems such as AUTOMATH however, strong normalization (and hence normalization) does hold. Much work concerning (strong) normalization has been done by logicans studying systems of *natural deduction* and functional interpretations (cf. for instance [5], [8], [10]). Their methods often apply to AUTOMATH also. Some new proofs of normalization and strong normalization have been given by members of the AUTOMATH-project (cf. [9]).

6.3.3. Church-Rosser theorem; uniqueness of normal forms

Question 6.3.1ii) above amounts to the *Church-Rosser theorem*: *If* A = B *then* A ≥ C ≤ B *for some* C. An alternative formulation of this is the *Diamond property* for ≥: *If* A ≥ B *and* A ≥ C *then* B ≥ D ≤ C *for some* D (cf. figure).



Diamond property

As a corollary of the Church-Rosser theorem we mention the *uniqueness of normal forms*: *If* B *and* C *are normal forms of* A *then* B ≡ C. This property together with the normalization theorem allows us to speak of *the* normal form NF(A) - computable by an effective procedure NF - of correct expressions A. The Church-Rosser theorem holds in the full λ-calculus as well as in typed systems. In AUTOMATH languages without η-reduction the standard λ-calculus proofs simply carry over (cf. [9]). In fact, in view of strong normalization, a slightly easier proof can be given here. For, e.g., AUT-QE, where we

have η-reduction the proof is somewhat more complicated and depends heavily on the closure theorem. The author intends to publish this proof and the other proofs omitted in this section in his doctoral dissertation.

6.3.4. Closure property

Let us first formulate the *closure theorem*: *If* $\mathcal{B}; \sigma \vdash A$ (*respectively* $\mathcal{B}; \sigma \vdash A \underline{E} B$) *and* $A \geq C$ (*with respect to* $\mathcal{B}$) *then* $\mathcal{B}; \sigma \vdash C$ (*respectively* $\mathcal{B}; \sigma \vdash C \underline{E} B$). In connection with the closure theorem, which holds for AUT-QE, we have two important derived rules:

1) *General substitution principle* (as mentioned in 2.5): *If* $x_1 \underline{E} \Sigma_1, \ldots, x_k \underline{E} \Sigma_k \vdash B$ (resp. $\vdash B \underline{E} C$) *and* $\sigma \vdash A_i \underline{E} \Sigma_i^*$ (for $i = 1, \ldots, k$) *then* $\sigma \vdash B^*$ (resp. $\vdash B^* \underline{E} C^*$), where Σ^* stands for $[\![x_1, \ldots, x_k / A_1, \ldots, A_k]\!]\Sigma$.
2) The *"left-hand equality rule"* (compare with the rule of type-conversion, which is the "right-hand equality rule"):
If $\vdash^{(3)} A \underline{E} B$ *and* $\vdash A = C$ *then* $\vdash C \underline{E} B$.
For 2-expression A we only have a weaker version in view of type-inclusion: *If* $\vdash^{(2)} A \underline{E} B$ *and* $\vdash A = C$ *and* $\vdash^{(2)} C \underline{E} D$ *then* $\vdash C \underline{E} B$ *or* $\vdash A \underline{E} D$.

6.4. A decision procedure

6.4.1. Deciding =- formulas

Suppose A and B are correct expressions. The normal form procedure NF (section 6.3.2) easily yields a decision method for the equation A = B, namely A = B iff NF(A) ≡ NF(B). Often, however, it is not necessary to compute normal forms for deciding A = B. For example, when A and B have different degrees one can easily draw a negative conclusion. Or more important, it generally happens that a few well-chosen reduction steps in A or B will result in a non-normal common reduct. The choice of efficient reduction steps here is a matter of *strategy*; the termination of a procedure which successively applies reduction rules to A or B is anyhow guaranteed by the strong normalization property, no matter in what order the reduction steps are applied. In order to prove the correspondence between decision procedure and language definition we must know that all the expressions in the reduction sequences from A and B to some common reduct are correct again. This is indeed the case by the closure theorem.

6.4.2. Deciding $\underline{E}$-formulas and expressions

6.4.2.1. Assume $\mathcal{B}$ is a correct book and σ a correct context; we must define a decision procedure for the correctness of $\underline{E}$-formulas and expressions. It will appear that this problem can be reduced to the decision problem for =- formulas (but for the straightforward task of checking the validity of the identifiers used).

6.4.2.2. Uniqueness of types

We know (by the rule of type conversion) that for all B' with $\vdash B = B'$ we have $\vdash A \underline{E} B \Leftrightarrow \vdash A \underline{E} B'$.

For 3-expressions A the converse (*uniqueness of types**) holds too:

(*) $\quad \vdash A \underline{E} B$ *and* $\vdash A \underline{E} B' \Rightarrow \vdash B = B'$.

For 2-expressions A we must be somewhat more precise in view of type-inclusion. We define among the correct expressions the relation $\subseteq$ by:

i) $[x_1,\alpha_1] \ldots [x_k,\alpha_k][y,\beta]\underline{\text{type}} \subset [x_1,\alpha_1] \ldots [x_k,\alpha_k]\underline{\text{type}}$

ii) $[x_1,\alpha_1] \ldots [x_k,\alpha_k][y,\beta]\underline{\text{prop}} \subset [x_1,\alpha_1] \ldots [x_k,\alpha_k]\underline{\text{prop}}$

iii) $\subseteq$ is the transitive closure of = and $\subset$.

Then instead of (*) for 2-expressions A we can prove

$$\vdash^{(2)} A \underline{E} B \text{ and } \vdash^{(2)} A \underline{E} B' \Rightarrow \vdash B \subseteq B' \text{ or } \vdash B' \subseteq B .$$

6.4.2.3. Now assume that A is correct. Then we can define a "*mechanical type*" *function* CAT, such that:

i) $\vdash^{(3)} A \underline{E} B \Leftrightarrow \vdash^{(3)} A$, $\vdash B$ and $\vdash \text{CAT}(A) = B$

ii) $\vdash^{(2)} A \underline{E} B \Leftrightarrow \vdash^{(2)} A$, $\vdash B$ and $\vdash \text{CAT}(A) \subseteq B$.

So CAT computes some canonical representative of the class of B' with $\vdash A \underline{E} B'$; furthermore, this B' is minimal with respect to $\subseteq$. For the actual definition of CAT we refer to [11, section 7]. Since the decision procedure $\overset{D}{=}$ for equations in the current checker also contains the possibility of type-inclusion - i.e. $A \overset{D}{=} B$ iff $A \subseteq B$ - the type function CAT reduces the verification of $\underline{E}$-formulas to the verification of equations.

*) Here we mean uniqueness with respect to definitional equality (=), in contrast with section 6.3.3, where we mean uniqueness with respect to syntactic equality ($\equiv$).

6.4.2.4. Finally we point out a decision procedure for correctness of expressions. Here we proceed by induction on the length of expressions. As an example we treat the case of application expressions <A>B where A and B are already supposed to be correct.

6.2.4.5. Uniqueness of domains

For function-like expressions A we define α to be the *domain* of A if

$$\vdash A \underline{E} [x,\alpha]\Sigma \quad or \quad \vdash^{(1)} A = [x,\alpha]\Sigma \ .$$

For domains we have *uniqueness* also (by the closure theorem and the Church-Rosser theorem): *If* α *and* β *are domains of* A *then* $\alpha = \beta$. This fact allows us to speak about *the* domain of function-like expressions. Now we are able to define a "*mechanical domain*" *function* DOM (for which we refer to [11, section 7]), which for function-like A picks out a canonical representative of the domain of A. The termination of DOM(A) follows by induction on the degree of A, using strong normalization.

6.2.4.6. By CAT and DOM the verification of correctness of <A>B reduces to the verification of some suitable equation: $\vdash$<A>B $\leftrightarrow \vdash$A *and* $\vdash$B *and* $\vdash$ A $\underline{E}$ DOM(B) or, equivalently, by 6.4.2.3i),

$$\vdash \text{<A>B} \leftrightarrow \vdash A \text{ and } \vdash B \text{ and } \vdash \text{CAT}(A) = \text{DOM}(B) \ .$$

6.2.4.7. For the other cases of correctness of expressions we refer to Zandleven again. The correspondence of the current verifier with the actual language definition is either immediate or follows from the above facts about CAT and DOM.

7. References

[1] De Bruijn, N.G.; *The mathematical language AUTOMATH, its usage and some of its extensions.* Symposium on Automatic Demonstration (Versailles December 1968), Lecture Notes in Mathematics, Vol. 125, pp. 29-61, Springer-Verlag, Berlin, 1970.

[2] De Bruijn, N.G.; *Automath, a language for mathematics;* notes (prepared by B. Fawcett) of a series of lectures in the Séminaire de Mathématiques Supérieures, Université de Montréal, 1971.

[3] De Bruijn, N.G.; *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem,* Indag. Math., 34, No. 5, 1972.

[4] De Bruijn, N.G.; *The AUTOMATH Mathematics Checking Project,* this volume.

[5] Girard, J.Y.; *Interpretation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur,* Doctoral dissertation, Université Paris VII, 1972.

[6] Howard, W.A.; *The formulae-as-types notion of construction,* unpublished 1969.

[7] Jutting, L.S. van Benthem; *The development of a text in AUT-QE,* this volume.

[8] Martin-Löf, P.; *An intuionistic theory of types,* unpublished 1972.

[9] Nederpelt, R.P.; *Strong normalization in a typed lambda-calculus with lambda-structured types,* Doctoral dissertation, Technological University, Eindhoven, 1972.

[10] Prawitz, D.; *Ideas and results in proof theory,* in: Proc. 2nd. Scandinavian Logic Symp., North-Holland Publ. Comp., Amsterdam, 1971.

[11] Zandleven, I.; *Verifying program for AUTOMATH,* this volume.