"Mathematics is what can be represented in Automath": a controversial proposition, what it means and what it implies

M. Randall Holmes

April 15, 2020

1 Introduction

This is an extended essay on the proposition expressed in the title. The proposition needs to be explained (the reader may not know what Automath is), evidence for and against it considered, and this proposition compared with other statements of the form "Mathematics is...".

It is not a proposition that we were initially inclined toward. We are in practice admirers of the usual foundations of mathematics in Zermelo-style set theory, and have spent a lot of our time on study of the alternative foundations in systems related to Quine's "New Foundations". We have spent time working on systems of automated reasoning, but our previous projects have involved implementation of systems related to "New Foundations" (and their very standard prerequisites in first-order logic).

Automath was the first substantial software designed for computer verification of mathematics. It was developed by de Bruijn and associates in the 1970's. There is a modern implementation, due to Freek Wiedijk, but it is not itself a system now in active use. Some computer-assisted reasoning systems, such as Coq, can be identified as intellectual successors of Automath.

The basis for the claim in the title is a view that the philosophical underpinnings of Automath in fact capture what mathematicians do and what they study. Part of what is needed to be able to understand and evaluate this claim is to exhibit what the philosophical underpinnings of Automath

are. This is aided by the fact that the core ideas of Automath are compactly surveyable.

In Automath all mathematical objects are typed. Though by temperament we are inclined to prefer the untyped world of ZFC or New Foundations, we do recognize the reality that we think of mathematical objects as being of various sorts.

Among the types is a type `prop` of propositions, and further, for each proposition p, a type of proofs of p, which is in fact identified notationally with the proposition p itself (in Automath in its original form, but not as we will see in our favored implementation of these concepts, where we use the notation `that` p for the sort of object (proofs of p) associated with the object (proposition) p).

This leads to the phenomenon usually called the "Curry-Howard isomorphism", the observation that the constructions of types in certain systems of types of mathematical objects correspond more or less exactly to the logical constructions of propositions from other propositions. This encourages the idea that a mathematical proof is itself a mathematical object. We present some typical identifications of logical constructions with constructions of mathematical types. A proof of $p \wedge q$ can be taken to be a pair consisting of a proof of p and a proof of q, so conjunction of propositions (if they are identified with the associated types) is a special case of the cartesian product. A proof of $p \rightarrow q$ can be taken to be a function taking proofs of p to proofs of q: so implication is correlated with the construction of simple function types. A proof of $(\forall x : \tau.\phi(x))$ can be taken to be a function taking a object x of type τ to a proof of $\phi(x)$: this is a function of a dependent type (the type of the second input depends on the type of the first).

The Curry-Howard isomorphism is most often considered in connection with constructive logic, but it can be used in connection with classical reasoning and indeed Automath was historically used to implement theories based on classical logic, though it could be used equally well to implement constructive theories and its successor Coq is primarily devoted to constructive reasoning (though one can implement classical reasoning in Coq as well).

The central logical engine of Automath is actually simply a type checker for dependent function types (which does double duty, as of course it serves to type-check mathematical objects which proofs talk about as well as serving to check the validity of proofs themselves). Part of understanding our proposition is understanding how a type checker for dependent types can even *be* the central logical engine of a proof checker. At this point we should

probably proceed to a concrete description of Automath sufficient to ground our discussion.

2 Description of Automath

This section contains a description of Automath. Automath admits various notational variants (discussed by Freek Wiedijk in the documentation of his implementation of Automath): we choose one without explicit deliberation.

2.1 Automath books and lines

An Automath text (called a *book*) consists of *lines*. Each line has four components, a *context*, an *identifier*, a *definition* and a *type*.

The format of a line is

$$\texttt{context} * \texttt{identifier} := \texttt{definition} : \texttt{type}$$

These are not necessarily best considered in the traditional order.

The identifier component of a line gives us the name of an object or operation being postulated or defined.

The context is typographically either the special symbol `0` or the name of an object (a variable) but this is a clever covert notation for a list of arguments. `0` represents an empty argument list. Whenever a line of the form

$$\texttt{C} * \texttt{I} := \texttt{EB} : \texttt{T}$$

is executed, a new context is created, whose name is `I` (not to be confused with the object otherwise represented by `I`) which consists of the argument list `C` with the new argument `I`, a variable of type `T`, appended. It is important to notice that the type `T` may depend on the variables in the context `C`. The letter `I` refers to this new context only when it appears as a context item in a line; otherwise of course it refers to the variable `I`. The use of the special string `EB` as the definition component indicates the special function of this kind of line.

A variable is in scope only if it appears in the current context, and new variables with the same names and different types can be declared and used in subsequent lines in contexts which do not include the original variable. A context with a given variable name refers back to the most recent declaration of a variable with that name (this could be made more sophisticated using the Automath paragraph system, but we prefer to keep things simple).

In our implementation described in a subsequent section, we prefer to give argument lists in full, and in fact we give them after the identifier, giving a more familiar format for a function definition, but the Automath context notation is rather clever and we will not evade presenting it here.

Variable names may of course appear with different meanings in different contexts. We do not allow any other multiple uses of identifiers in the version of Automath we are describing (we do not implement the paragraph system).

Primitive objects or operations are postulated using lines of the form

$$\texttt{C} * \texttt{I} := \texttt{PN} : \texttt{T}$$

where the special string `PN` (for "primitive notion", we suspect) indicates the special function of the line. This declares a new constant (if C is `0`) or operation taking arguments of the types indicated by the context `C` and producing an output of the indicated type `T` (which may itself depend on the variables appearing in the context `C`). The identifier `I` may be used in a term of the Automath language either by itself or with an argument list (a parenthesized list of terms separated by commas) of length not exceeding that of `C`. It *always* implicitly has arguments of the types indicated by considering the context `C` in which it was defined: a bit of Automath cleverness is that where the context of the line in which a term headed by the operator `I` has initial segment identical to an initial segment of the context `C` in which the identifier was declared (matching types as well as names of identifiers) the initial arguments may be omitted.

A primitive notion, once postulated, is always in scope (unless its identifier is overwritten): the only effect of context on its use is the optional shortening of argument lists indicated. In the version of Automath described here, we do not allow the declaration of a constant object or operation whose name coincides with a variable previously used.

Lines of the form

$$\texttt{C} * \texttt{I} := \texttt{D} : \texttt{T}$$

where `D` is a term of the Automath language, define an operation named `I` (or a constant if `C` is `0`), with the same syntactical privileges as the operation which would be introduced by the line with `PN` in place of `D`, just in case the term `D` actually has the type `T` (this is where the type checking comes in!). And of course (abusing notation, thinking of `C` as a complete argument list not abbreviated by its final letter) we are postulating the definition `I(C) =`

`D`: the operation `I`, followed by any argument list with types matching those of the context `C` in which it was defined, will give the result of replacing the variables in `C` with the actual arguments supplied.

As above, we do not allow reuse of names.

We briefly outline a more liberal management of namespace. Automath allows the opening and closing of *paragraphs*. The identifier of the current paragraph is a list of distinct strings. The command `+p` appends the string `p` to this list. A technicality: to reopen a paragraph already closed, `+*p` must be used. I suppose that just `+p` overwrites the previously saved paragraph. The command `-p` deletes `p` and all following strings from the list (closing many paragraphs at once, potentially). Each identifier is then supposed to have a suffix of the form "`p1... .pn`" (the names of the paragraphs it is embedded in, separated by dots, and enclosed in double quotes). To refer to an identifier not in the current paragraph, attach an explicit suffix to the identifier to which an initial segment agreeing with an initial segment of the locally appropriate suffix can be prepended to give its full suffix. There can be only one such suffix, since all paragraph names in any particular suffix must be distinct. Of course local identifiers (or ones whose full suffix is an initial segment of the current one) need no suffix at all. Identifiers in contexts may not have explicit paragraph suffixes. This paragraph system would in effect allow names to be reused in local contexts [This is a note to myself against a possible implementation: my impression from the sources is that the behavior of the paragraph system was variable.]

A fine point is that in the paragraph system, a variable name in the current context will be read as that variable, overriding any other definition of the same symbol which might make sense in the current paragraph.

In an implementation, my intention would be to allow many notational variants described by Wiedijk.

2.2 Automath terms and types

As is the case with Automath syntax, there is a menu of choices of type systems found in various versions of Automath. We choose one without explicit deliberation.

Nontrivial Automath terms appear in lines in the definition and type slots.

An Automath term may be of one of the following shapes:

primitive constant: `TYPE` and `PROP` are primitive constants.

variable: A variable must either be meaningful in the context of the current line or bound in the term.

construction: An instance of a defined construction will consist of a declared identifier `I` either by itself (if it is a constant or if its arguments are exactly an initial segment of the context) or followed by an argument list as in `I(term1,...,termN)`.

application: If `t` and `F` are terms, `<t>F` is a term (notice that this is $F(t)$: de Bruijn chose to write application in the reverse order).

abstraction: If `x` is a variable, `t` is a term standing for a type, and `T` is a term, `[x,t]T` is a term. This is typed λ-abstraction $(\lambda x : t)(T)$.

Not all Automath terms are meaningful. Some are meaningful as objects (can appear as definition components of lines), some are meaningful as types (can appear as type components of lines) and some can do both.

The primitive constants `TYPE` and `PROP` can appear as types but do not themselves have types. The class of terms with these characteristics is closed under λ-abstraction: if `T` is a term which is a type and does not have a type, and `t` is a term which is a type and has a type, then `[x,t]T` is a term which is a type and does not have a type. All such terms are constructed in this way. These are the items which can appear only as types and not as objects.

The terms which can appear as types and themselves also have type have type of one of the previous kinds. There is subtyping: any term which is a type and has type `[x,t]T` also has type `T`. This is the only subtyping in Automath. These are the items which can appear both as objects and as types.

A variable must be assigned a type which itself has a type. A construction (with or without parameters) can be postulated or defined with type of its output a type which does not have a type: its inputs must be of types which do have types.

A term `I(term1,...,termN)`, where `I` was declared in context `C`, is well-typed if we first pad the argument list with an initial segment of the current context so that it has the same length as the context `C = (C1,...,CM)`, and moreover this initial segment is identical to the corresponding initial segment of `C`, then attempt type matching of the list of types `(c1,...,cM)` of the identifiers `Ci` with the list of types `(tau1,...,tauM)` of the terms `termi` (renumbered to start with the initial padding), where the definition of type matching is that the types of `c1` and `tau1` are the same, and the types of the list `(c2[term1/C1],...,cn[term1/C1])` obtained by replacing the terms `C1` where it occurs in the types of subsequent identifiers in the context with `term1`, match the types `(tau2,...,tauM)` in the same sense. The null argument list of course matches only the null argument list. The type of this term will be the result of replacing each `Ci` with `termi` in the type assigned to the operation `I` when it was declared.

If `t` has type `tau` and `F` has type `[x,tau]T`, `<t>F` will have type `T[t/x]`, the result of replacing the variable `x` with `t` in `T`.

If a term `T` is assigned type `tau` when a variable `x` possibly appearing in it is assigned type `t` (the type `tau` might depend on `x` too), the term `[x,t]T` is assigned type `[x,t]tau` (which can be understood as `t -> tau` if `x` does not occur in `tau`).. Such a term is only well-typed when `t` is a term which can be a type and has a type.

Certain terms can be reduced to other terms. Terms which reduce to the same term are regarded as identical for purposes of type checking.

If `I(term1,...,termN)` is well typed and `D` is the definition component of the line declaring `I`, this term reduces to the result of padding the argument list to `(term1,...,termM)` as above then replacing each `termi` with `Ci` in `D`, where `(C1,...,CM)` is the context in which `I` was declared.

`<t>[x,tau]T` reduces to `T[t/x]`. This is the usual beta-reduction. `[x,tau]<x>F` reduces to simply `F`.

Issues with the correct definition of substitution and with identification of terms arise because of the presence of variable binding. We stipulate that in fact every term `[x,tau]T` is to be read as

$$\texttt{v\{tau,T[v/x]\}[T[v\{tau,T[v/x]\}/x]}$$

where `v` is a unique variable symbol which we never actually use, and each `v{tau,T[v/x]}` is another atomic variable (certainly substitutions have no effect on it). In other words, every bound variable has an actual name other than the one we see which is uniquely determined by its context: this renaming should be supposed to have taken place before any substitution into a λ-term, and terms which differ only by renaming of bound variables are in fact to be understood as identical. De Bruijn introduced de Bruijn indices in order to solve these issues!

Both types and object terms can be reduced.

3 Doing logic in Automath

4 Description of Lestrade

5 Automath and logicism

6 Automath and ZFC-ism

7 Automath and data type abstraction