The use of the language Automath for syntax and semantics of programming languages
de Bruijn, N.G.

Published: 01/01/1975

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 27. Dec. 2018
The use of the language AUTOMATH for syntax and semantics of programming languages.

by N.G. de Bruijn
Department of Mathematics
Eindhoven University of Technology
Eindhoven, The Netherlands
Telephone (0)40 - 472807

Abstract.

Mathematical theories can be written in the form of books in the language AUTOMATH, and the checking of the correctness of the entire text can be automatized.

The idea of this paper is to consider syntax and semantics of computer programming languages as parts of mathematics and to write semantical theorems concerning particular programs in books that already contain logic, mathematics, definition of syntax, and axioms for semantics of the programming language. The paper describes ways for doing this, with a programming language that is a kind of subset of ALGOL 60 without being much less powerful.
1 Introduction.

AUTOMATH ([1,3,4]) is a language which permits us to write very large parts of mathematics in such a precise way that verification of the correctness of the mathematical contents can be carried out by a computer. The AUTOMATH book that is presented to the computer has to contain everything: logical foundations, inference rules, mathematical foundations, axioms, definitions, formulas, abbreviations, theorems, proofs, and the mutual connections between all these things.

Since the beginning of the AUTOMATH project in 1968 a considerable amount of mathematical material has been written and checked, and therefore it seems to be feasible to start the use of AUTOMATH for practical purposes.

In many areas of mathematics one may feel that there is not much need for such extremely precise formulations of complete theories. Such areas can have a strong intuitive background, and the feel of safety is supported by the many applications that can be checked by various entirely different methods leading all to the same result.

There are fields, however, where the need for precision is very strong, where intuitive support is weak and experimental evidence misleading. This may happen in cases of long proofs consisting of very many elementary steps, where it is strongly felt that a chain is as weak as its weakest link. If the number of steps runs into thousands, the need for mechanical verification can be a very practical one. In such cases it may be worth while to go into the trouble of coding every detail into a final formalization, removing all traces of intuition and experience.

One such field can be the one that is devoted to proving that a computer program has the semantics we want it to have, i.e. that program execution does what we claim it to do. Intuition and experimental evidence
in this field are known to be independable. One reason, but not the only one, is that long programs are often produced in cooperation between several programmers, pieces are taken from libraries, etc. Another matter is that program languages themselves need a thorough formal description, both for syntax and semantics.

Let us think of complex situations where logic, mathematics, syntax and semantics are interwoven, and where it is just the interplay between these components that requires attention. Imagine we have a program for searching the solutions of a number-theoretical problem, and we wish to prove that the program does what we claim. As it happens, during the search all sorts of shortcuts are made, partly on the basis of number-theoretical or combinatorial theorems, partly on the basis of semantical knowledge about parts of the program. In the future we might even imagine that we are working in an advanced programming language that allows us to extend the syntax and semantics in the course of the argument, and to prove semantical theorems on these extensions that are used in the proof of the final semantical statement on the total program (i.e. the statement that execution of the program produces exactly all solutions of the number-theoretical problem). In a situation like this the supervision over the whole system seems to be inadequate if we produce in the "human" way, consisting of a patchwork of formalized and intuitive pieces, tied together by our feeling of confidence. Usually the situation is less complex than we just described, but at least some of these difficulties are always present. Part of the difficulties are not so much sources of errors, but yet produce a kind of conceptual uneasiness since we feel unable to lay a connection, other than by intuition, between the formalized mathematical notion in the mathematics text and its formalized counterpart in the program.
It is just this kind of a need for an overall survey of a large field, that AUTOMATH was devised for.

In order to be able to talk in less general terms, we act as if there were only one interesting program in the world, i.e. a program for finding the g.c.d. of two integers. We start from an AUTOMATH book that contains an amount of basic material. We can extend the book by adding further material (every acceptable addition to an acceptable book produces a new acceptable book). The book certainly has to contain

(i) Logical tools, inference rules.
(ii) Mathematical foundations, in particular properties of natural numbers and integers.
(iii) Definition of the g.c.d., and proofs of some of its properties.

We now add a chapter to the book, involving

(iv) A definition of a particular programming language.
(v) Basic assumptions on the semantics of programs in that language.

Next we write in the book

(vi) A description of a program for the computation of the g.c.d.
(vii) A semantic theorem, with proof, about that program (this theorem may state something to the effect that if the input is m,n, then the output is the g.c.d. of m and n).

However complete this may seem to be, yet there are a number of things that have to be taken for granted. These things concern the basic assumptions on the system as a whole, and not such details as described in (iii), (vi) or (vii).

2. What has to be taken for granted?

In the first place we have to believe the things clearly indicated as "primitive notions" in the AUTOMATH book itself. Some of them refer to primitive objects for which no definition is given: we just give them a name and we say of what type they are. Others
refer to axioms, which are things that have the form of a theorem without proof.

The way AUTOMATH deals with assertions is what is now called the "formulas as types" presentation of logic (the term comes from Howard [7]). Objects have a type (the type of 3 is "natural number", the type of S is "set"); proofs have a type too: the type of a proof a the (unique) assertion it proves. So corresponding to a "definition" (showing (a) name of the object; (b) description of its construction; (c) type of the object) we have a "theorem" (showing (a) name of the proof; (b) construction of the proof; (c) the assertion). Corresponding to the introduction of a primitive object we have an axiom: in both cases (b) is replaced by "PN" which is just a warning symbol. There is a third kind of lines in our book: the block opening lines; they may differ from the PN lines in the sense that they are narrowing the context: They can be "let x be an object of type A" or "let p be a proof of assertion B". They embody what we call introduction of a local variable or the introduction of a local assumption. They lose their power as soon as we get back to the original wider context.

As it is the case with any formal system, there is an unformalized system of interpretation that goes along with it. For the primitives in the formal text we choose interpretations in the outside world, and we have a system of propagation of interpretation that produces an interpretation of the final results in the formal text. We need not bother about interpretation of the intermediate parts of the text. (If the final result is a theorem, the thing we care to interpret is the assertion and not the proof). Interpretational troubles are restricted to primitives and finals, and this is exactly the reason for the practical use of formal systems.

Let us direct our attention to the book that checks the g.c.d. program. We have to do various things. First the formal soundness of the AUTOMATH
text has to be believed. This has to be based on the belief that definition and theory of AUTOMATH are sound, and that the text checking algorithm refuses every input that is not according to the language definitions (actually this fact is an example of a quite complex semantical theorem). And we have to believe that the computer language that is used for describing that checking algorithm is adequately implemented on a computer that makes no mistakes.

Once the AUTOMATH text has been taken as absolutely sound, we still have the interpretations. We have to interpret the notions and axioms of (i) and (ii), and to make sure they correspond to ideas in the partly formalized mathematical world we think we live in. This has to remain vague since that mathematical world is a bit vague. The only thing we can say here is that this correspondence can be traced by careful thinking in the same style as mathematicians always have been thinking intuitively about the meaning of their symbolisms. And experience reinforces confidence.

Next we turn to (iv). The program primitives and constructs described in the AUTOMATH book, will not be exactly the same as the programs we really give to our computer. There will be an algorithm that syntactically translates one into the other, and we have to believe that this is sound.

The more important thing is (v). After we have interpreted the semantic axioms we see that they can be considered as conditions that have to be satisfied by the computer system we want to present our g.c.d. program to. To convince ourselves that it does satisfy these conditions, is partly a semantical, partly an engineering problem.

The final interpretation of (vii) is very simple. There will not be any difficulty in reading its main result as "the g.c.d. algorithm evaluates the g.c.d.".
Use of Types.

We want to describe syntax and semantics of an ALGOL-like programming language in terms of AUTOMATH. The first thing we have to decide on is which things are types and which things have types.

In AUTOMATH there are 3-expressions, 2-expressions, and the single 1-expression type. Every 3-expression has a unique type, and that type is a 2-expression. Every 2-expression has type as its type. We shall use 2-expressions for some fundamental classes of mathematical objects, and 3-expressions for those objects themselves. Also, we use 3-expressions for proofs, and their types are the assertions these proofs prove. In order to easily build assertions, we create as a primitive notion the 2-expression "bool" and, again as primitive notion, we build, for every 3-expression b of type bool, the 2-expression TRUE(bool). If we have a 3-expression p of type TRUE(b), the interpretation is that p is a proof for the truth of the proposition b. From now on the interpretation of logic and mathematics is rather straightforward.

In order to express programs, we start with taking 2-expressions Ω, interpreted as state spaces. If Ω is a type, we create the primitive notion "program (Ω)" as a 2-expression. The interpretation of a 3-expression π having type "program (Ω)" is that π is a program acting on the state space Ω.

Syntax.

In order to describe the syntax of a programming language, we shall introduce primitive programs and primitive program constructs. On the basis of these we are able to construct all programs of the language, but it does not seem to be necessary to state an axiom that every object of type program (Ω) belongs to this constructed set.

In order to give a vague idea without going into details of AUTOMATH
we write a few lines of our book:

\[
\begin{align*}
\text{boo.} & := \text{PN type} \\
\text{b} & := \text{bool} \\
\text{TRUE} & := \text{PN type} \\
\Omega & := \text{type} \\
\text{program} & := \text{PN type}
\end{align*}
\]

The horizontal lines indicate that b and Ω are variables (or "block openers"), the vertical lines indicate blocks with a common context (the context of the line with TRUE is b). Note that on the basis of this, "program (Γ)" can be used for every possible 2-expression Γ, in every possible context. For example we can say a thing like: "let f be a function that attaches a program on Ω to every integer n" (whence n is a variable the program depends on, and not a variable the program acts on), or "let f be a function that attaches a program on Ω to every program on Ω." 

We shall certainly need the notion of a cartesian product of two state spaces, and therefore we write

\[
\begin{align*}
\Omega_1 & := \text{type} \\
\Omega_2 & := \text{type} \\
\text{cartprod} & := \text{PN type}
\end{align*}
\]

Let us now discuss how to introduce primitive programs. We may create programs on every state space as well as programs on special state spaces (in ALGOL'60 the first case does not occur). For example one might create the empty program (to be interpreted as a program whose execution does not alter the state), and one might create fake programs, like a program of which we give no semantic information whatsoever.
A very important group of primitive programs are assignments. A clear description of them is by no means easy. The simplest cases are the replacements $x := y$ where $x$ and $y$ are state space variables. It can be a case where the state space is $\text{cartprod}(\mathbb{R}, \mathbb{R})$; then the replacement can be introduced as a primitive program (the interpretation is, if we use the terminology of analytic geometry, horizontal projection onto the diagonal, and its semantics has to be arranged accordingly. (Needless to say, it seems unattractive to write separate axioms for cartesian products of $2, 3, \ldots$ spaces; it is better to introduce operations on finite sequences of spaces, which is easier to do if we use the extension AUT-QE of AUTOMATH).

A harder question is what to decide about value assignments like $x := c$, where $c$ is a variable the program depends on, and $x := 0$, $x := 1$, etc. At first sight it seems reasonable to postulate (if $x$ is a state space variable of the type integer) that "for every integer $c$ the statement "$x := c$" is a program. But one should be aware of the fact that this is said in a mathematical language that allows to substitute other expressions for $c$, and in particular these might be expressions a computer cannot evaluate. One should realize that the postulate "$x := c$ is a program for every integer $c$ in the range $0 \leq c \leq 1$" has a stronger expressive power than the two postulates "$x := 0$ is a program" and "$x := 1$ is a program" have together, unless we add extra axioms in the latter case that create the same effect (like axioms on "definition by cases"; the formulation of such axioms seems to require "equality of programs" as a primitive).

It seems to be safe to be very restrictive about value assignments. In the case of integers one could just take "$x := 0$" and $x := 1$" (as soon as we have addition as a primitive program, we can simulate an assignment like "$x := 216$" by a program having no other assignments than "$x := 0$" and "$x := 1$").
On the other hand, if one studies mathematical algorithms instead of computer programs, there is no objection against admitting "$x := c$".

In the case that $x$ is a state space variable of the type "bool", we can take "$x := \text{true}$" and "$x := \text{false}$" as assignments (true and false are primitives of type bool, for which we have the axioms of classical logic in the logico-mathematical part of our book). Moreover we can add a few others like: if $x, y, z$ are state space variables of type bool, real, real, respectively, then "$x := y > z", "x := y = z"$ etc. are programs. And, in analogy to the addition of integers, we can use primitives like conjunction, implication, etc., in order to get programs that simulate the more complicated boolean expressions.

A special kind of assignments is connected with arrays. For some special data types $\Lambda$ we allow the use of assignment operations on variables of type $[n, Z] \Lambda$ (i.e. the type of all mappings of the set of integers $Z$ into $\Lambda$). If $f$ is a variable of this type, if $x$ is a variable of type $\Lambda$, and $k$ a variable of type $Z$, we have to postulate the primitive programs whose interpretation is $f[k] := x$ and $x := f[k]$. The treatment of arrays is hard if array bounds are variables, as it is possible in ALGOL'60. We shall not discuss this in this paper (it should be remarked that its treatment in [6] is unsatisfactory since it neglects the fact that the variables that presented the array bounds may change inside the block; a correct way to describe how such variables are untouchable might involve the introduction of variables that cannot be assigned to syntactically).

Thus far we discussed primitive programs; next we get to primitive program constructs, i.e. ways to compose bigger programs from smaller ones. First the concatenation (first $\pi_1$, then $\pi_2$):
A construct (not existing in ALGOL'60) that has the same syntactic structure as the concatenation, is the disjunction \( \pi_1 \text{ or } \pi_2 \) (where the programmer leaves it to the machine which one to execute).

A very important primitive construct is the binary selection \( \text{"if } b \text{ then } \pi_1 \text{ else } \pi_2 \". We want to restrict this to the case that \( b \) is a state space variable of type bool (otherwise we would have to describe in our book that \( b \) is a syntactically admissible boolean expression, and that seems to be hard). Trusting the simulative power of a primitive language, we might limit ourselves to the case that \( b \) does not occur in the state space \( \Omega \) of \( \pi_1 \) and \( \pi_2 \), and that \( b \) is the variable that corresponds to the extension of \( \Omega \) to \( \Omega \times \text{bool} \). In our book this looks like this:

Next we consider the constructs "projection" and "injection" that transform programs in one state space into programs in another state space. We shall be a bit superficial here, since a correct treatment requires elaborate indexing of the various components.

Imagine \( \pi \) is a program on \( \Omega = Z \), where \( \Omega \) is any state space and \( Z \) stands for the set of all integers. Now we want to consider the "projection" on \( \Omega \), i.e. the program on \( \Omega \) that would be described in ALGOL'60 as
begin integer n; n end.

In our m... we introduce this primitive as follows:

\[
\begin{align*}
\Omega &:= \text{type} \\
\pi &:= \text{program (cartprod(\Omega, \Omega))} \\
\text{proj} &:= \text{PN program(\Omega)}.
\end{align*}
\]

The notion of injection is a case of a procedure called by name. Let \( \pi \) be a program on a state space \( \Omega_1 \times \cdots \times \Omega_k \), and let \( x_1, \ldots, x_k \) be the names of the variables for these \( k \) components. Let \( \phi \) be some one-to-one mapping of \( \{1, \ldots, k\} \) into \( \{1, \ldots, m\} \), and let \( \Lambda_1 \times \cdots \Lambda_m \) satisfy \( \Lambda_\phi(i) = \Omega_i \) \( (i = 1, \ldots, k) \). Then we want to introduce the program "injection \( \phi(\pi) \)" on \( \Lambda_1 \times \cdots \Lambda_m \) whose interpretation is as follows: take \( y_1, \ldots, y_m \) as state space variables for \( \Lambda_1 \times \cdots \Lambda_m \). Replacing in the program \( \pi \) every instant of \( x_i \) by \( y_\phi(i) \) gives the program "injection \( \phi(\pi) \)."

The constructs described thus far are mappings of programs or pairs of programs onto programs. Let us call them "lower" constructs in contrast to "higher" constructs where the variables are not programs, but objects on a higher level.

The only example we take is this one: If \( Q \) is a function that maps programs onto programs, then the primitive higher construct "recurs(\( Q\))" is to be interpreted as the recursively defined procedure "\( n := Q(\pi) \)". In our book we get

\[
\begin{align*}
Q &:= \text{type} \\
Q &:= [\pi, \text{program}(\Omega)] \text{program}(\Omega) \\
\text{recurs} &:= \text{PN program(\Omega)}
\end{align*}
\]

The syntactic elements presented in this section seem to cover only a
small fragment of ALGOL'60. Nevertheless quite a large part of ALGOL'60 can be simulated by this fragment, in particular much of the practice concerning procedures called by value or by name, and function procedures. For some details of such a simulation we refer to [6].

5. Semantics.

As our point of view we select relational semantics.

This means that semantic results on a program π (with state space Ω) are given in the form of a binary relation on Ω. If we say that π satisfies the relation P then the interpretation is as follows: for every pair \( w, w' \in Ω \) which are such that \( w \) and \( w' \) can be the initial and final value of the state, \( P(w,w') \) is true (but we do not require the converse). We shall also express this by saying that P presents information on π. We do not require this information to be as complete as possible: we just try to work with P's which have a simple form, as long as they are adequate. One of the advantages of this kind of semantics over Floyd's inductive assertion method is that we state results about the program and not about the program plus a number of assertions scattered over it. Our relational semantics is still pragmatic: just like in any mathematical proof, we never bother about intermediate steps being as strong as possible, it is only the final result that counts. And relational semantics has not the slightest trouble with non-deterministic programs.

It is possible to extend the state space by improper elements standing for non-determination and abortion (see [2],[6]). We shall not try to describe that here, since it is a bit awkward in connection with the cartesian product structure of our state spaces. A compromise that would make things only slightly harder, is to express semantic information by means of a predicate Q and a relation R, with the following interpretation: if the initial state \( w \) satisfies Q(\( w \)), then the program execution terminates, and the final state satisfies R(\( w,w' \)). (See [6]).
In our AUTOMATH book we formulate relational semantics by means of a single primitive $w$. We take it to be a 3-expression, and the interpretation is a proposition. (Any 3-expression that has $\text{TRUE}(w)$ as its type is a proof for our semantic assertion). The variables are $\Omega$ (a state space), $\pi$ (a program) and $P$ (a relation on $\Omega$). The text is

\[
w := \text{type} \\
\pi := \text{program} \\
P := \{w, \Omega \mid [\omega', \Omega]\} \text{bool} \\
w := \text{FN} \text{ bool}
\]

As a general axiom on $w$ we can formulate that for all $\pi, P_1, P_2$ we have (if $\Omega$ is fixed we write $w(\pi, P)$ instead of $w(\Omega, \pi, P)$):

\[
w(\pi, P_1 \wedge P_2) \iff w(\pi, P_1) \land w(\pi, P_2).
\]

This implies monotonicity (if $P_1 \supset P_2$ then $w(\pi, P_1) \supset w(\pi, P_2)$). One might formulate this for infinite conjunctions too, but that seems to be unnecessary.

For every primitive program and for every primitive construct we have to formulate a special semantic axiom. At least for the primitive programs and lower primitive constructs these axioms come rather naturally, and there is no point where it is a problem how to write the axiom in AUTOMATH once we feel what it should express. We give a few examples. First the axiom for the concatenation. It says that if $P_1, P_2$ are relations (state space $\Omega$), if $P_1$ presents information on the program $\pi_1$ and $P_2$ on $\pi_2$, then the "boolean matrix product" $P_1 \times P_2$ presents information on the concatenation of $\pi_1$ and $\pi_2$. (The boolean matrix product is

\[
(P_1 \times P_2)(\omega, \omega') = \bigvee_{\rho \in \Omega} \{ P_1(\omega, \rho) \wedge P_2(\rho, \omega') \};
\]
For the binary selection we have the following. If $\omega, \beta$ are used as coordinates in cartprod ($\Pi, \text{bool}$), if $P_1$ and $P_2$ present information on $\pi_1$ and $\pi_2$, then $P_3$ presents information on "if b then $\pi_1$ else $\pi_2$" (or in the terminology of our book: binselect($\Pi, \pi_1, \pi_2$)), where $P_3(\omega, \beta, \omega', \beta')$ is defined as

$$(\beta = \beta') \land ((P = \text{true}) \rightarrow P_1(\omega, \beta, \omega', \beta')) \land ((P = \text{false}) \rightarrow P_2(\omega, \beta, \omega', \beta')).$$

In order to describe the semantics of $\text{recurs}(Q)$, where $Q$ is a program-to-program function, we start from the fake program $\pi_0$ (for which we assume nothing but $w(\pi_0, \text{TRUE})$, where TRUE is the identically true relation). We now build the programs $Q(\pi_0), Q^2(\pi_0) (= Q(Q(\pi_0))), Q^3(\pi_0), \ldots$. Let $P_1, P_2, \ldots$ be relations such that (for $n = 1, 2, \ldots$) $P_n$ presents information on $Q^n(\pi_0)$. Let $P$ be such that for every pair $\omega, \omega'$ there exists an $n$ such that $P_n(\omega, \omega') \rightarrow P(\omega, \omega')$. Then $P$ presents information on $\text{recurs}(Q)$. In this pragmatic formulation we avoid talking about things like Scott's minimal fix point (for which we refer to [21]) thus leaving the way open for non-deterministic cases.

After having described the principles of syntax and semantics in our AUTOMATH book, we can proceed by proving general lemma's, e.g. on the syntax and semantics of the "while do" statement. From there on we can start describing special programs and their semantics, building up a library of results in the same way as an AUTOMATH mathematic book builds up its own inference rules, mathematical foundations, lemma's, theorems, up to the final results whose correctness we want to check. It is a long way to go, but the fact that nothing needs to be done twice, provided the setting has been chosen with sufficient generality, may help us through.

AUTOMATH can be used in various ways for logic and mathematics (cf [5], section 14); with syntax and semantics we have the same situation. The difference between different ways of using the language lies mainly in the use we make of the typing operation. One way to live with it is to restrict its use to a minimum, thus clinching as close as possible to standard predicate calculus plus Zermelo-Fraenkel set theory. But we can also use typing in a more liberal fashion, to the effect that the intuitive idea that things have types, is reflected in the language. The present paper has been written in this liberal spirit. The effect is that AUTOMATH's machinery of definitional equality can be used freely for the handling of programs, but on the other hand it cannot be refuted any more. This means that, whether we want it or not, definitionally equal programs have the same semantics.

As long as this concerns so-called delta-reduction (i.e. the effect of replacing things by their definition) this is what everybody wants. On the other hand, if we have the $\beta$- and $\eta$-reduction of the $\lambda$-calculus, we hardly know what it means when dealing with programs. In particular we have the question why two programs equivalent by $\beta$- and $\eta$-reduction, should necessarily have the same semantics. The answer is probably this: As long as the syntactic and semantic axioms have a particular form (not involving $\lambda$-expressions) then it can be proved that if two programs (not involving $\lambda$-operations on the type "program") are definitionally equivalent, this equivalence can be established already without $\lambda$-operations on the type "program". A proof of this might establish the fact that our liberal attitude is not just effective but also innocent.
References.


