Manual for the MANIAC Automath Checker

M. Randall Holmes

4/5/2018: added discussion of theories supported

This is the manual for the MANIAC implementation of Automath. MANIAC is a recursive acronym for "MANIAC: a nice interactive Automath checker".

In spite of its name, it is probably not very nice. But it is interactive, which has not been the case for any previous implementation of Automath to my knowledge. The only available modern implementation is the `aut` implementation by Freek Wiedijk which is batch but infinitely faster. The `aut` program checks Jutting's translation of Landau in less than a second; MANIAC takes about 3 hours to check this file, alas. However, MANIAC is generally quite suitable for the purpose of typing Automath lines at your terminal and seeing what happens. It can also be noted that MANIAC implements a somewhat stronger logic, though I think that upgrading `aut` to the MANIAC logic would likely be trivial.

Again, `aut` does not support the extension of the Automath language with single-quoted strings used as prefix and infix operators, as far as I know. This extension is described in the yellow Automath book but as far as I know was never implemented in any actual Automath checker, and my choice of APL precedence conventions for prefix and infix operators, while natural for Automath notation, does not seem to be found in the Sources.

Contents

1 The Automath language (basics)

In this section we describe the Automath language as a medium for expressing mathematics.

Identifiers are any strings of upper case and lower case letters and numerals, possibly followed by one of $, !, # followed by a numeral, or ## followed by a numeral. The latter four kinds of identifier are special identifiers generated by the system. The identifiers with `#` are renamed bound variables generated by the system; the others are explained below.

There are two reserved identifiers, `prop` and `type`. The strings `ERROR` and `PaNiC` should probably not be used as identifiers since their appearance in MANIAC output would normally signal an error reported by the parser or display functions.

There are three additional term constructions, abstractions, function applications, and constructions.

1. Where x is an identifier used as a (bound) variable and A and T are terms, $[x : A]T$ is an abstraction term (x is bound in the subterm T but not the subterm A).

2. Where f and a are terms, $\langle a \rangle f$ is a term (representing application of the function f to the argument a; the reversal of the traditional order of application does have merits and in any case must be supported to read existing Automath text).

3. Where f is an identifier declared as a construction with arity $\geq n$ and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)$ is a construction term: f by itself is a construction term as well in this case.

Things mentioned in an Automath text fall into three species, entities, sorts, and metasorts. Each entity or sort has a type. The type of an entity is a sort. The minimal type of a sort is a metasort: there is a subtyping relation on metasorts, so a sort actually may have multiple types, but each sort has a unique metasort minimal in the subtype order.

The reserved identifiers `prop` and `type` are metasort terms. If M is a metasort term and A is a sort or metasort term, and x is an identifier, $[x : A]M$ is a metasort term. All metasort terms are either of this form or are application terms which reduce to this form (there are no facilities to define abbreviations of metasort terms as there are for terms of the other

species). The subtype relation can be defined at this point. $[x : A]\texttt{prop}$ is a subtype of `prop` when A is a sort; $[x : A]\texttt{type}$ is a subtype of `type` when A is a sort; if M is a subtype of N then $[x : A]M$ is a subtype of $[x : A]N$; computationally equivalent terms are subtypes of one another, and the subtype relation is reflexive and transitive. All subtype relations between metasorts are determined by these conditions.

An abstraction term $[x : A]T$ is of the same species (entity, sort or metasort) as T. The term A must be a sort or metasort (in classic Automath, it would be restricted to being a sort). If T has type U, $[x : A]T$ has type $[x : A]U$. The variable x is a variable bound in the subterm T (but not in the subterm A) of type A.

An application term $\langle a\rangle\, f$ must have the domain of f equivalent to a type of a [not necessarily its minimal type] (and so, since a must have a type, it must be of the species entity or sort, in classic Automath always an entity). The domain of f can be computed by computing the minimal type of f or the minimal type of the type of f, one of which which will be reducible to a metasort term $[x : A]T$: A must also be the type of a. If the type of f is U, the type of $\langle a\rangle\, f$ will be $\langle a\rangle\, U$ (which may be reduced if U is an abstraction term).

A construction term $f(t_1, \ldots, t_n)$ actually abbreviates a term $\langle t_n\rangle \ldots \langle t_1\rangle\, f!$ where $f!$ is an abstraction term associated with f whose relation to f will be explained below. Since it abbreviates a term built with operations already discussed, its typing conditions have already been indicated implicitly. Each construction has an arity. If f has arity n, it may appear with no arguments or any number of arguments less than n, as well as with all n arguments given explicitly: in this case the term is read as missing an initial segment of its argument list which is replaced with an initial segment of the current context (a concept to be explained below), as long as these variables taken from the context have appropriate type. In classic Automath this would be done only if the initial segment of the current context coincided with the corresponding initial segment of the context in which the construction was defined, but we do not make this restriction. The most striking effect of this is that f as an atomic term does not stand for the function $f!$ that is applied to the argument list of a construction term with leading f, but to the generic value of $f!$ with parameters taken from the current context (in classic Automath, always the generic value with the exact parameters with which it was defined).

We indicate the definition of the notion of substitution of a term a for a

variable x in a term T, represented by $T[a/x]$. $v[a/x]$ for a variable v is a if $x = v$ and v otherwise. $(\langle A\rangle\, T)[a/x]$ is $\langle A[a/x]\rangle\, T[a/x]$. $([y : A]T)[a/x]$ is $[u : A]T[u/y][a/x]$ where u is any variable distinct from y and not appearing in T or a or where $u = y$ and y does not appear in a (all choices of u give equivalent terms). $f(t_1, \ldots, t_n)[a/x] = f(t_1[a/x], \ldots, t_n[a/x])$. Here where we discuss appearance of a variable in a term we mean simple typographic occurrence.

We do not claim it as a virtue, but MANIAC does not follow the practice (originating in the Automath project!) of using de Bruijn indices to handle bound variables; for our sins, we actually use named bound variables in the internals, and MANIAC includes in its internals the implementation of substitution outlined in the previous paragraph. MANIAC does include (not one but two!) conventions for renaming bound variables in abstraction terms already constructed to give standardized or pretty-printable forms. Internal representations may include large integer suffixes to variable names to ensure freshness, though the user would not normally see these.

We now consider how to show that terms are equivalent (this is essential for being able to determine when application terms (and so construction terms) are well-formed: the exact application is to show that terms representing types are equivalent (or that one stands for a subtype of the referent of another), to verify that an application can be carried out). Equivalence is reflexive, symmetric, and transitive. If A is equivalent to A', then $T[A/x]$ is equivalent to $T[A'/x]$ for any choice of T and x. $[x : A]T$ is equivalent to $[y : A]T[y/x]$ when y does not occur in T. $[x : A]\,\langle x\rangle\, T$ is equivalent to T when T does not include x. Finally and crucially, $\langle a\rangle\,[x : A]T$ is equivalent to $T[a/x]$ in all cases. Equivalences are only asserted between well-formed and well-typed terms. Note that construction terms are expanded using the rule for applications of abstractions just exhibited (of course first filling in any missing arguments). Equivalence of terms is computable by a process of reduction using the stated rules.

It should be noted that our term language and type system are richer than those in any previously implemented dialect of Automath because we allow abstraction over sort variables and application to sort terms. We did this not so much to enrich the type system (though we have done so by allowing this) as to simplify the implementation of constructions, which have always been allowed to take sort arguments, but which could not be conveniently implemented as values of appropriate abstractions because the related abstractions were not legal abstractions in the original systems.

MANIAC notation has been enhanced to support infixes. Arbitrary strings enclosed in single quotes are now an additional pool of identifiers, which are recognized by the parser and display functions as infixes (or prefixes). When supplied with two or more arguments (whether input in infix format or not) they will be displayed in infix notation. The parser will accept any construction as an infix, but will require more parentheses to support use of an ordinary identifier as an infix operator. The order of operations is the old APL order: any term to the left of an infix other than an atomic identifier must be parenthesized; parentheses to the right of the infix are optional. A term input in infix form with a non-quoted construction will be displayed in the usual form. Arguments are omitted as usual in both parse and display in infix terms (which may cause them not to be infix!). This actually supports mixfix: what is happening is that the argument list of any construction can be partitioned into two lists and the constructor placed between them. In the display, only quoted identifiers will be shown as infixes, and they will always appear after the first argument. Use of quoted identifiers as prefix operators is also supported, with the same precedence conventions (which may be disconcerting, as one might expect prefixes to bind more tightly than infixes even when all have equal precedence, and this is not the case).

2 The Automath context and Automath lines

The Automath environment consists of the context and the paragraph system. The paragraph system is described in a subsequent section.

The context is a list of variables and associated types. No variables can occur in the type of a variable in the context which are not either variables appearing earlier in the context or bound variables. A context has a conventional name. An empty context has the name 00 (this is our notation; in older dialects of Automath the empty context has been named 0 or referred to using the empty string, and the style of reference in the Jutting translation, which uses no name for the empty context at all (or may be thought of as using the null string as its name), is supported). The formal name of a nonempty context is the name of its last variable.

It is worth noting that while parsing the term T in a term $[x : A]T$, the context is extended with x of type A. This has an effect on the process of supplying omitted arguments. In the display functions, this extension of the context does not happen, so arguments supplied during the parsing process in this way will appear in the displayed form of a term.

An Automath line has four parts, the name of a context, an identifier being declared, a value being assigned to that variable (or reserved words replacing the value in the case of lines with other functions) and a type to be assigned to that value.

If the value part is replaced with the reserved word `EB` (the yellow Automath book says that this abbreviates "empty block opener": it has notational variants), the effect of the line is to change the context by appending a new variable (with shape given by the identifier part of the line and type given by the type part of the line) to the context named in the context part, and setting this to be the current context (as well as declaring the identifier to stand for that context in the current paragraph).

If the value part is replaced with the reserved word `PN` (this abbreviates "primitive notion": it has notational variants), the effect of the line is to declare a primitive construction with argument list defined by the context part of the line and type defined by the type part of the line. The type part should contain no variable not either bound or found in the context (in case the variable is found in the context, the construction is seen to have type dependent on an argument).

If the value part is an Automath term, this term (and the type term) may contain no variable not either bound or found in the context, and the

effect of the line is to define the construction named by the identifier as the function taking parameters with the form of the context to the value term (and having type equal to the type term: MANIAC will check that the type term is a type of the value term).

A construction declaration, whether it is a definition or the declaration of a primitive, is in effect saved in the form $f(x_1 : \tau_1, \ldots, x_n : \tau_n) = V : T$ (where V may be `PN` if this is primitive). A term of the form $f(t_1, \ldots, t_n)$, where f is defined rather than primitive, will be expanded to the form

$$\langle t_n \rangle \ldots \langle t_1 \rangle \, [x_1 : \tau_1] \ldots [x_n : \tau_n] V$$

and reduced (with attendant type checking) for evaluation; its type will in any case (whether the construction is primitive or defined) be reported as $\langle t_n \rangle \ldots \langle t_1 \rangle \, [x_1 : \tau_i] \ldots [x_n : \tau_n] T$ (and appropriately reduced, with attendant type checking).

The abstraction $f!$ mentioned earlier has now appeared in its explicit form $[x_1 : \tau_1] \ldots [x_n : \tau_n] V$. Its type $[x_1 : \tau_i] \ldots [x_n : \tau_n] T$ has the conventional name $f\$$.

Notice that since our constructions, whether primitive or defined, must have values with types, their values must be entities or sorts: they must belong to the species which have types. We do not provide the facility to define metasort constants or named functions with metasort values. We could do this, but it is not a move that we anticipate making.

3 Talking to MANIAC

MANIAC is a Python program. I edit the source in IDLE and type commands at the Python prompt. One can go into a MANIAC interface by typing `interface0()` at the Python prompt.

One can do three things at the `interface0()` prompt. One may type `quit` to exit. One may type a comment (a line beginning with `#`; if it begins with `##` it will be preceded by an additional blank line in the display). A comment will be posted to the log file (the default log file is `automathlog.aut1`, to which all commands and MANIAC's replies will be posted as well as to the screen; comments can actually be useful later). Finally, one can type an Automath line.

The most basic style of typing an Automath line to MANIAC is to write four terms separated by spaces, which are simply the context, identifier, value and type parts of the line in that order. The special value terms PN and EB must be followed by whitespace to avoid confusion with variables starting with these sequences of letters. This style is my own dialect: it is not supported in the Sources.

MANIAC supports some additional special options for some of the parts which are principally useful in an interactive context. The identifier part may be replaced by _, in which case no declaration is actually made, though type checking happens. The type part may be replaced by `??`, in which case MANIAC computes and reports the type (so one may define a notion and have MANIAC compute the type to be recorded).

There are alternative ways to present Automath notation and lines which are supported.

The special value EB may also appear as eb (which must be followed by a space) or `---`.

The special value PN may also appear as pn (which must be followed by a space) or `'prim'` (quotes are part of the notation!) or `???`.

The reserved metasort terms `prim` and `type` may also appear as `'prim'` and `'type'`, respectively.

The context part of a line may be closed with `@` or `*`, and in this case no whitespace is required between the context and identifier parts. If a line starts with `@` or `*` then the context part is understood to be 00.

The context part of a line may be omitted, in which case the context of the previous line in the same paragraph is used.

A line declaring a variable x of type A may take the form $[x : A]$. Several

lines of this form may appear on one line of text, possibly preceded by a context part.

The symbol `:=` may separate the identifier and value parts of a line. The symbol : may separate the value and type parts of a line.

The additional features listed support the style of the Grundlagen, in which no whitespace at all appears in lines. We do not support all of the variant styles supported by `aut`, it must be reported. In particular, we do not support any style in which the order of the four parts of a line are permuted (though we do support the `[x:A]` style in which only two parts of the line appear, and multiple lines of this form may appear together on the same line of text).

MANIAC will reply in various ways. When a context is successfully extended with a new variable, it will be exhibited as a list of variables with types. When a function is declared, its declaration or definition will be given in a format including the context as a list of typed formal parameters, roughly

$$f(x_1 : \tau_1, \ldots, x_n : \tau_n) = V : T$$

MANIAC also isses error messages. Any lines from MANIAC start with `%`, which is the sign of a transient comment (it will not persist if the log file is executed by MANIAC, unlike a comment starting with `#`, which will be copied.)

MANIAC can execute log files. The command

`readbook`(`filename1`, `filename2`)

will execute commands in `filename1.aut1` and copy the lines and MANIAC replies to `filename2.aut2` (which will itself be a valid log file if all goes well). The command `readbook2(filename1,filename2)` reads `filename1.aut2` to `filename2.aut1`.

A MANIAC session at the interface ends with `quit` and a MANIAC file to be read by one of the `readbook` commands should end with a line consisting of just `quit`, or it will hang when run.

4 Name management and the paragraph system

Names can be redeclared in Automath generally, and under MANIAC, with some abandon. Nothing blocks declaration of variables conflicting with construction names or bound variables conflicting with constructor names. MANIAC will not allow you to declare a constructor with the name of a variable in the current context. It will allow you to bind a variable with the same name as one in the current context (which is added to the context during the parsing process and supersedes the earlier item with the same name temporarily). If a toggle is set to do this, MANIAC will warn the user when an identifier meaningful in the current paragraph or the most recently closed paragraph is redefined, but this does not create an error condition: the user is just advised of this and the redeclaration goes ahead.

No construction declaration is ever lost, however masked: construction f declared on line 234 can always be invoked as `f.234`. If one displays a term with this f in it and it cannot be interpreted correctly in the current context as just f, it will be displayed as `f.234`. Single-quoted identifiers with line number qualifications are not treated as prefix or infix operators by the parser or display functions: they will appear applied to argument lists in the same way as any other construction. This is our own contribution: classic Automath has no such feature.

The interpretation of names is managed by the further device of paragraphs. It should be noted that our implementation of the paragraph system is not identical to the original implementation by Zandleven, although it is close enough that MANIAC successfully runs the Jutting translation without any changes to its text. At any point in an Automath text, one is in a nested sequence of paragraphs. The topmost one is called `top`. The long name of the current paragraph is formed as the string of all local names of the nested paragraphs to which one belongs (starting at the top) separated by dots. Whenever a construction or context is declared, it is declared qualified by the long name of the current paragraph (and constructions are separately declared qualified by the current line number).

A context or construction can appear qualified with a paragraph name (possibly a string of local paragraph names separated by dots, which may optionally include an initial dot) enclosed in double quotes. MANIAC will attempt to interpret a context or construction name qualified by a dot ini-

tial paragraph name by extending the long name of the current paragraph with it and seeking the name thus qualified. It will attempt to interpret a context or construction name qualified by a non-dot-initial paragraph name (or without any paragraph qualification) by appending that paragraph name (or the null string if there is no paragraph qualification) to the long names of each paragraph in which the current context is contained, starting with the smallest one (that is, with the one with the longest name) and looking for a declaration of the name in question thus qualified (choosing the first one it encounters). A variable cannot be qualified with a paragraph name: a variable name so qualified must be the context part of a line. We further refine this by allowing the checker to search the most recently closed paragraph as a last resort in interpreting a name. This is not the same as the original specification, but it does work with the Grundlagen. The original specification required that the first local name in a paragraph qualification actually be the local name of a paragraph containing the one in which one is currently working (except in the case of a dot-initial paragraph name); we do not require this, as it allows us to avoid actually parsing paragraph names, and this works in practice.

A new paragraph is opened with a line `+paraname` (`paraname` being the local name of the new paragraph). A paragraph is closed with a line `-paraname`. A paragraph is re-opened with a line `+*paraname`. When a paragraph is opened, the current context is saved, so that it can be restored when that paragraph is closed. MANIAC currently draws no distinction between re-opening a paragraph and opening a new paragraph with the same name: there is some information that it stores differently (information about locally declared constructions whose declarations perhaps ought not to be changed) which for the moment is not used, as MANIAC does not even block you from redefining notation used in the current paragraph.

Here are Python functions which allow the user to view declarations:

Showconstruction: Takes one string argument. Will show you the locally relevant declaration of the identifier argument. Currently returns a Python error if the identifier is not declared: I'll try to remember to fix this.

GetConstructions: Takes one string argument. Returns all declarations of this identifier, qualified appropriately by paragraph references and line numbers.

I should add support for these to the interface (note to self).

5 Optional Settings and Theories Supported

The software supports various options. We list ones here which users might actually want.

`ToggleConstructionTypes()` turns on or off the ability to have sort variables bound in abstractions. We see no particular reason to turn this off. It is interesting to observe that even when this toggle is turned to the off position, the internals of the checker still manipulate terms with abstractions with bound sort variables in the course of definitional expansion and type checking of construction terms: but the user is constrained from writing such terms and can expect not to see them. Turning this off gives AUT-QE more or less exactly.

`ToggleNoSubtyping()` turns on or off the general phenomenon of subtyping metasorts. By default this is on.

`ToggleInsaneTypes()` turns on or off a stronger definition of subtyping under which any $[x : A]$`prop` is a subtype of `prop` and any $[x : A]$`type` is a subtype of `type`. We conjecture that the resulting highly impredicative theory may be consistent, but we have not verified this. The theory with the original subtyping of AUT-QE has straightforward semantics describable in set theory.

`ToggleAllUniversals()` turns on or off strong subtyping for propositional types only. By default this is now on.

`ToggleDisplay()` turns on or off the display of arguments of constructions which Automath normally allows itself to omit. There is no effect on the parser, which handles omitted arguments in any case.

`ToggleShowType()` will turn on (or off) the display of types computed by MANIAC.

`ToggleExperimental()` turns on or off a feature which causes additional declarations to be computed. When this toggle is on, when a construction f is computed the construction $f!$ (described above) of the abstraction associated with f and the construction $f\$$ of its type function (if it has one) will be declared as well. If `ToggleConstructionTypes()` is in the off position, fewer additional declarations will appear.

`ToggleShowMatches()` turns on or off a diagostic setting which causes MANIAC to display matches that it is attempting.

`ToggleWarnings()` turns on warnings about redeclarations of identifiers meaningful in the current or the just closed paragraph. No errors are thrown in these situations: the user is simply advised.

There are some other toggle functions, but they are ones we do not envision a user wanting to mess with.

We indicate how various flavors of Automath are implemented. Some are new.

AUT-68: There is no particular setting for this. Set toggles as for AUT-QE, and use no metasort but `type` (don't use the primitive `prop`, and use no abstraction metasorts), and you will be working in AUT-68.

AUT-QE: Run `ToggleConstructionTypes()` (this is on by default; it needs to be off for AUT-QE).

AUT-QE-NT: Run `ToggleConstructionTypes()` and `ToggleNoSubtyping()` to get AUT-QE with no subtyping.

AUT-QE2: This is our nonce name for the default state of MANIAC. This is usable for the purposes of AUT-SL (we have yet to implement the actual construction of terms from books, but we shall). AUT-QE2-NT (obtainable by running `ToggleNoSubtyping()`) could also be used and would be closer to de Bruijn's original proposal of AUT-SL. All versions here differ from de Bruijn's in not having any term of degree 4: we see no reason to assign types to metasorts.

The insane extension: Run `ToggleInsaneTypes()` and you get a theory of whose status I am uncertain. With regard to the subtyping enabled by the on setting of `ToggleAllUniversals()`, we are fairly certain that this is *consistent*: we wonder whether it might allow proof irrelevance to be demonstrated (we conjecture not, but we are not certain). If one is worried about this, one can turn it off to get a theory intermediate between AUT-QE and the default state AUT-QE2 of MANIAC, in which universal quantification is not automatically present over sorts not legal in AUT-QE.