Help with Automath

Randall Holmes

March 2, 2018

Automath is intended to be simple...it is intended to be serviceable as a basic framework for mathematics, so it ought to be!

The world of Automath consists of entities, sorts and metasorts. Each entity has a type, which is a sort. Each sort has a type, which is a metasort.

In the beginning, all we are given are two basic metasorts, the metasort `prop` inhabited by propositions, and the metasort `type` inhabited by types. A proposition is thought of as a sort, inhabited by proofs of that proposition (or evidence for that proposition). A type is a sort inhabited by ordinary mathematical objects. So entities are either proofs or objects.

The Automath world is divided into a logic part (the propositions and their proofs) and an ontology part (the types and the mathematical objects which inhabit them).

We can build functions freely with domains any sort and ranges rather arbitrary. If x is an entity variable, A is a sort, and T is an entity, sort, or metasort term (probably containing x) then $[x : A]T$ [which you might think of as $(x : A \mapsto T)$ or $(\lambda x : A : T)$] is an entity, sort, or metasort term (it is of the same one of these three species as T). If T is an entity or sort term it has a sort or metasort type U (which also might depend on x), and the type of $[x : A]T$ is $[x : A]U$.

Flipping this around, if f is a function taking entities x to $f(x)$ of type $U(x)$, then the type of f is $[x : A]U(x)$. If $f(x)$ is an entity [in which case f is itself an entity], then $U(x)$ and so $[x : A]U(x)$ will be a sort, as expected. If f is a sort, then $[x : A]U(x)$ will be a metasort as expected.

Notice that our identification of a type of a (dependently typed) function f with the function taking elements x of the domain of f to the type of $f(x)$ means that we are not identifying types as sets containing the things of that

type. When we wish to speak of the collection of things of a type, we will try to talk about the *extension* of that type.

We make our first remark about the subtyping on metasorts (which we will say more about presently). Every sort is either of metasort `prop` or metasort `type`. There are other metasorts which sorts can have, but they are subtypes of these two. There is a subtype order on metasorts, and the type in the strict sense of a sort is its minimal metasort in the subtype order. This allows us to describe the logic side and the ontology side of things: every entity has a (unique) type which is a sort which has metasort `prop` (in which case the entity is a proof) or metasort `type` (in which case the entity is a standard mathematical object). A function of the form $[x : A]T$ where T (which may depend on the variable x) is either a proof, or a proposition, or a metasort included in `prop` (the logic side) or a mathematical object, a type, or a metasort included in `type` (the ontology side). The typing of T determines which side we are on. A can be on the ontology side or the logic side (a proposition or a type) independently of which side T is on.

We describe a logical operation which happens to be definable strictly in terms of the Automath primitives, and which will help us to see how the subtyping works. $p \to q$ (the usual implication operation on propositions) is defined as the sort inhabited by functions taking proofs of p to proofs of q. If I have a function f which takes any pp (a proof of q) to a proof of q (suppose we can write it as $T(pp)$) then the function f can be written $[pp : p]T(pp)$ and its sort will be $[pp : p]q$ (q being the sort of $T(pp)$ independently of what pp is). Now the metasort of $[pp : p]q$ is $[pp : p]\texttt{prop}$. Since we do want an implication to be a proposition, we must suppose that $[pp : p]\texttt{prop}$ is included in `prop`.

Similarly, if we have an object type N (say, the natural numbers), and a predicate P of objects of type N (which would be a function from elements of N to propositions, and so a sort of type $[n : N]\texttt{prop}$, and in fact a species of proposition, we would expect a proof of $(\forall n \in N : P(n))$ to be a function taking inhabitants n of N to proofs of $P(n)$. Perhaps confusingly, the type of such a function is P, so in fact the type of proofs of $(\forall x \in N : P)$ is identified with the function P from N to `prop` itself! The universal quantifier $\forall$ turns out to be as it were the identity function on the sort $[n : N]\texttt{prop}$. Once again, a universally quantified proposition is regarded as a proposition, so we expect $[n : N]\texttt{prop}$ to be a subtype of `prop`.

We briefly describe the system of metasorts. `prop` is a metasort. `type` is a metasort. For any metasort M and entity sort A, $[x : A]M$ is a metasort.

Notice that the entity sort A might depend on entity variables, which allows for dependent typing in the formation of further metasorts. The subtyping relation works as follows. Every metasort is constructed by adding finitely many prefixes to either `prop` or `type` (so each metasort is a subtype of one of these two basic metasorts). Each $[x : A]$`prop` is a subtype of `prop`, each $[x : A]$`type` is a subtype of `type`, $[x : A]M$ is a subtype of $[x : A]N$ just in case M is a subtype of N, and there is no subtyping not determined by these relationships. Each sort will have a minimal type in the subtype order, with a prefix as long as possible.

In this way we see that the full system of metasorts is determined as soon as we know what sorts there are and how a sort may depend on an entity variable.

To enrich the type system to the point where we can do useful mathematics, we need to give an account of how to declare and define constructions in an Automath theory.

In our first pass, we will consider Automath lines as taking the usual form of parameterized function definitions $f(x_1, \ldots, x_n) = T$.

We further decorate such a declaration with information about types: $f(x_1 : \tau_1, \ldots, x_n : \tau_n) = T : U$.

We further note that each τ_i (understood to be the type of the parameter x_i) is a sort or metasort term. A term τ_i may depend on parameters x_j for $j < i$. T may be an entity or sort term, or the special term `PN`, and U will be a sort or metasort term (expected to be the sort of T): both T and U may of course depend on any of the parameters. If T takes the shape `PN`, the declaration is of a primitive construction f taking parameters of the indicated types to a result of type U, which can be freely used subsequently. Otherwise the new identifier f is being defined as a function in a way familiar from mathematics of quite an elementary level: the definition will succeed if the given typing is correct.

Automath lines as they are usually presented hinge on a clever device for building argument lists. Each line has four parts, the *context*, the *identifier*, the *value*, and the *type*. The context actually determines the list of variables in the function declaration being given. The special value 00 indicates an empty parameter list (appropriate for defining or declaring a constant). Any other value is a variable name, in fact the name of the last variable in the argument list of a previous line.

A line in which the value part takes the special value `EB` declares the identifier as a new variable of type given in the type part of the line to be

appended to the parameter list. Such a line simply extends the parameter list for use in a later line (and the identfier doubles both as the name of the new variable and the name of the new parameter list in which it is the last item).

A line in which the value part takes the special value `PN` declares the identifier as a new primitive construction which can be applied to lists of arguments of types matching those of the parameter list represented by the context part of the line, obtaining an output of type indicated by the type part of the line.

A line in which the value part is a term defines the identifier part as a function taking the parameter list represented by the context to the value indicated by the value part of the line, with the obligation (checked by the software) that the type part of the line really is a type of the value part.

The primitive functions of the Automath language are the function abstraction $[x : A]T$ described above, function application $\langle x \rangle f$ (yes, that is written backward, so that $\langle a \rangle [x : A]T$ is $T[a/x]$, if a has type A.), and primitive constructions $f(t_1, \ldots, t_n)$ introduced by definition and declaration lines as described above.

That is all there is to it. The interesting thing is that it is enough.

Notice that the constructions introduced by Automath lines can be thought of as functions themselves, but of types more general than are allowed for Automath notations. $f(x_1 : \tau_1, \ldots, x_n : \tau_n) = T : U$ might be thought of as introducing an object f of the form $[x_1 : \tau_1] \ldots [x_n : \tau_n]T$ with type $[x_1 : \tau_1] \ldots [x_n : \tau_n]U$, but notice that the x_i's may be sorts or entities and so the τ_i's are permitted to be metasorts as well as sorts. $f(t_1, \ldots, t_n)$ could then be thought of as abbreviating $\langle t_n \rangle \ldots \langle t_1 \rangle f$. But the forms of abstraction and application presented here are more general than Automath allows in its language.

In my Automath implementation, I actually evaluate constructions by building and reducing abstraction and application terms of these illegal forms, but the user does not see them. Automath allows constructions to appear only through their values. A construction f may appear with a shortened argument list – in this case, the argument list can be filled in from the front but only if the current context coincides exactly with the context in which f was defined. This is a clever scheme of abbreviation, best understood by seeing examples. In my implementation, the system will declare identifiers $f!$ and $f\$$ representing the actual f as a function and its type if they have legal Automath types. The constant f by itself, as in standard Automath, means

precisely $f(x_1, \ldots, x_n)$ in a context in which the first variables are exactly the parameters with which f was declared, with exactly the same types.