A Modest Proposal for the Semantics of Automath

Randall Holmes

February 22, 2018

We briefly describe the Automath universe we intend to represent. Vertically, the entities it considers are divided into three tiers, objects, sorts, and metasorts. Horizontally (and less importantly for our immediate purposes), it is partitioned into objects of logic (proofs, propositions, and metasorts of propositions, all subtypes of `prop`) and objects of the usual mathematical ontology (mathematical objects, their types, and metasorts of types, all subtypes of `type`).

Each object has a sort, and each sort has a metasort. The type of an object is its sort; the type of a sort is its metasort. We firmly halt the regress at that point.

Terms of the Automath language are of four kinds.

There are atomic terms (variables or declared constants) of each sort or metasort, and atomic metasort constants `prop` and `type`.

There are abstraction terms $[x : t]T$, where x is an object variable, t is a sort term, and T may be an object, sort, or metasort term. $[x : t]T$ will be of the same species (object, sort or metasort) as T, and the type of $[x : t]T$ will be $[x : T]U$ if U is the type of T. Renaming of bound variables does not change the meaning of an abstraction term.

There are function application terms $\langle x\rangle\, f$ (yes, this is written backward). The usual reduction of $\langle a\rangle\, [x : t]T$ to $T[a/x]$ works (in any species). If the type of f is $[x : t](U)$, the type of $\langle a\rangle\, f$ will be $\langle a\rangle\, [x : t](U) = U[a/x]$.

The other kind of Automath term is the construction term $f(t_1, \ldots, t_n)$. Constructions are introduced by declaration. The term $f(t_1, \ldots, t_n)$ can be read as meaning $\langle t_n\rangle \ldots \langle t_1\rangle\, f^*$, where f^* is an abstraction $[x_1 : \tau_1] \ldots [x_n : \tau_n]F$, with the added refinement that not only can F be of species either object or sort, but the types τ_i can be types of objects or sorts (and of course the types τ_i may depend on x_j's for $j < i$ and the type of F may depend on the x_i's). Constructions are not first class objects in Automath: they appear only through their values. One cannot abstract over constructions: particular ones are introduced by declaration in an Automath theory. [1]

Finally, we observe the detail that every metasort $[x : t]$`prop` is a subtype of `prop`, every metasort $[x : t]$`type` is a subtype of `type`, and if M is a metasort and a subtype of N, then $[x : t]M$ is a subtype of $[x : t]N$. This is the only subtyping in Automath, but it is quite weird enough on its own.

There have been complaints about the fact that the types of abstraction terms in Automath are abstraction terms. We believe this to be unwarranted. The point is that one should not be hypnotized by the idea that a type is the *set* of its instances. We take the view that an object with type is a pair whose second projection is its type. A function f from an object type τ to objects will then be implemented as a pair whose first projection is its graph in the usual sense (a set of ordered pairs) and whose second projection is a pair, which has as its first projection the graph of the map sending each x of sort τ to the sort of $f(x)$ and has as its second projection the graph of the map sending each x in τ to the minimal metasort of the sort of $f(x)$ in the subtype order.

We indicate how this entangled situation is created. Let κ be an inaccessible cardinal. We start by providing κ distinct base types of each size less than κ. To effect this, we provide that there are sorts $((\alpha, \beta), \texttt{prop})$ and $((\alpha, \beta), \texttt{type}$ for each $\alpha, \beta < \kappa$. Objects of sort $((\alpha, \beta), \texttt{prop})$ and $((\alpha, \beta)\texttt{type}$ are of the forms $(\gamma, ((\alpha, \beta), \texttt{prop}))$ and $(\gamma, ((\alpha, \beta), \texttt{type})$ where $\gamma < \alpha$. The parameter α is the size of a base sort, while β is as it were a serial number. Notice that empty sorts are supported! The objects representing the base metasorts `prop` and `type` can be chosen quite arbitrarily.

The model is constructed in a sequence of stages indexed by the inaccessible κ. At stage 0 we have just the base types described.

At limit stages we take unions. At a successor stage, we introduce new function types. To begin with, we introduce new metasorts as graphs of

[1]It is worth noting that in my implementation I actually represent construction terms as application terms involving functions of these illegal sorts and type and evaluate them by reducing these terms. I further provide a feature which automatically declares the functions (whether objects or sorts) which are thus identified with the constructions, and their types, if they do happen to be legal Automath objects and sorts: this allows a style in which use of bound variables can be minimized, which is of at least theoretical interest.

functions in the usual sense from the extensions of object types to metasorts already constructed. We have a restriction that metasorts are partitioned into levels. `prop` and `type` are level 0. A metasort of level $n+1$ is a function from the extension of an object sort to metasorts of level n. The subtype order on metasorts is defined as indicated above. We also have the restriction that each metasort has its range elements restricted to the proof side or the ontology side (metasorts which are either `prop` or functions with proposition metasort values, or on the other hand `type` or functions with type metasort values).

A sort s constructed at the new stage is a pair, with $\pi_1(s)$ the graph of a function from the extension of a type t constructed at an earlier stage into the set of all sorts constructed at earlier stages [the range must be restricted to the proof side or the mathematical object side]. $\pi_2(s)$ is then the function of minimal subtype order which sends each element of t to a metasort of $\pi_1(s)(t)$ (where every sort has as metasorts not only its second projection, but also all metasorts having its second projection as a subtype). That there is a uniquely determined such function is readily checked.

An object x constructed at the new stage is a pair, with $\pi_1(x)$ a function from a domain type t into the domain of all objects constructed at earlier stages [restricted to either the proof side or the ontology side], and $\pi_2(x)$ the object sort whose first component is the function taking each y of type t to the sort of $\pi_1(x)(y)$.

It is straightforward to prove by induction on κ that no more than κ objects, sorts and metasorts are constructed at any stage, and that no new object is introduced at stage κ. This relies quite strongly on κ being inaccessible! The Automath logic does not have the strength of an inaccessible: the reason this comes in is that we think of arbitrarily defined functions from object sort domains into the domains of all objects, all sorts, or all metasorts. Bare Automath does not have the power to define many such functions. However, the declaration facility of Automath can readily give constructions that can give lots of functions from object domains into the domain of all objects.

We can cut down the extravagant assumptions of this semantics. It is possible to postulate only a subset of the base types. For example, one could provide that all base propositional types have no more than one element, and get a quite adequate representation of proof objects with the sometimes convenient feature of proof irrelevance.

The inaccessible can be avoided. The ordinal κ can for example be taken to be any strong limit if one restricts the functions forming new entities from having range elements taken from stages with ordinal indices cofinal in κ. The strong limit strength is needed here if one presumes that function types are inhabited by *all* functions with the given domains. The reason that such restrictions are natural is that the facilities of bare Automath for defining functions from a type of objects into sorts are actually quite limited. But note that the ability to declare constructions in Automath (of types transcending the basic system!) makes it easy to formulate theories which require the full consistency strength underlying the model presented here, or even more.

This is what a logical framework should be like: it should have low logical strength in itself but enormous expressive power so that strong theories can easily be presented.

This can be contrasted with the metaphysics of Lestrade. Lestrade has as its object sorts (again partitioned into a proof side and an ontology side) the basic sorts `prop` of propositions and the basic type `type` of "type labels", and then basic sorts `that` p for each entity p of sort `prop` and `in` τ for each entity τ of sort `type` [we completely avoid identification of entities with sorts], and an additional basic sort `obj` provided to represent "untyped" mathematical objects. There are further abstraction sorts $(x_1 : \tau_1, \ldots, x_n : \tau_n) \Rightarrow \tau$, in which τ is an object sort, which may depend on any of the x_i's, and each τ_i is an object or function(!) sort, which may depend on x_j's for $j < i$. That is the entire Lestrade logical framework (apart from considerations related to rewriting); one proceeds further by postulating objects and constructions of these types (noting that whenever one postulates an object or method of constructing objects of sort `proof` or `type` one thereby extends the system of sorts). Lestrade is weaker in its metaphysical assumptions: notice that there is no provision for specific constructors for object types, even arrow types (on the ontology side) or the analogous implication types on the proof side: these are easily declared, but the point is that there is actual content to postulating them. This is an advantage for certain purposes: in Automath, as soon as one postulates a type one automatically has arrow types built from it, quantifiers over it, and even quantifiers over arrow types built from it. This will not happen in Lestrade without explicit postulation of constructions. Lestrade is stronger in its expressive power: there is no analogue to the situation in Automath where one cannot actually talk about the sorts of constructions which have sort inputs or outputs. Any construction one postulates in Lestrade has a function sort, and one can build constructions which take inputs of that function sort. Of course such a wealth of expressive power makes it very easy to express postulates which are very strong or inconsistent (which is also easy enough in Automath!)