A computer implemented philosophy of mathematics

M. Randall Holmes

May 14, 2018

This paper presents a philosophical view of the basic foundations of mathematics, which is implemented in actual computer software. The nature of the philosophical view is partially expressed in the fact that it can be implemented in actual computer software.

An important aspect of the view of mathematics that we are presenting is that the objects of mathematics are understood as being in a sense finite[1]. Though the mathematics we implement is classical and allows the implementation of ZFC and other theories of infinite sets, its native view is that all infinities are potential rather than actual. The way this is achieved is by viewing objects as (abstractions from) the expressions used to denote them [while not failing to notice the importance of the distinction between use and mention!] For some objects this requires no special comment (confusing natural numbers and the numerals used to represent them is harmless, as the sequence of numerals exhibits the formal properties required of an implementation of the natural numbers!) For functions (and sets) this does require comment. For us the notion of function is primitive (and "sets" are defined as functions[2]), and we view a function as a finitely presented object which can be applied to a (potential) infinity of diverse arguments of appropriate sorts. Again, the underlying view is that mathematical objects can be understood as abstracted from notation: the function $f(x) = x^2 + 1$ is abstracted from the expression $x^2 + 1$, in which the variable x is understood as the argument (this can be written more formally as $(x \mapsto x^2 + 1)$ or $(\lambda x : x^2 + 1)$). The

[1]which correlates well with "implementable on a computer".

[2]The same approach can be taken with the notion of set as primitive, by viewing sets (and relations) as abstracted from *open sentences*.

expression x^2+1 is a finitely given object which shows us how to compute the value of the function at any argument which may be presented to us [rather than an infinite table presenting all the values at once].

It is important to reiterate that in spite of this we will be able to implement ZFC and other theories of infinite sets, which rely on quite different intuitive pictures of what mathematical objects (and functions in particular) are like, without difficulty.

A second important aspect of the view of mathematics we take is that proofs are understood as mathematical objects. We make extensive use of (a version of) the Curry Howard isomorphism. We postulate a sort of propositions (called `prop`) and for each proposition p we postulate a sort `that` p inhabited by proofs of the proposition p (we sometimes prefer to refer to elements of `that` p as "evidence for p"). In usual presentations of the Curry-Howard isomorphism, a proof of $p \wedge q$ may be understood as an ordered pair consisting of a proof of p and a proof of q; a proof of $p \rightarrow q$ can be understood as a function from evidence for p to evidence for q. This is a well understood idea extensively developed in many other places. The Curry Howard isomorphism is usually taken as relating theorems of constructive logic to type constructions, but it is readily adaptable to classical logic.

More generally, mathematical objects in our scheme belong to sorts. (We say sorts rather than types because the name `type` has a special internal role in our framework, as we shall see shortly). The most popular foundations of mathematics are superficially untyped, but the use of types is ubiquitous in practical mathematics (even as implemented in the untyped ZFC) and the use of the Curry Howard isomorphism to implement logic means that the implementation of ZFC or other unsorted mathematical foundations in our framework actually involves the use of sorted objects. We provide a sort `obj` of "untyped mathematical objects" (the sets of an implementation of ZFC would be of sort `obj`) and a sort `type` of objects τ correlated with sorts `in` τ, each such sort being understood as inhabited by mathematical objects of a type labelled by the object τ. We will refer to objects of sort `type` as "type labels", though we may carelessly refer to them as "types" on occasion. When we are speaking carefully in terms of our own framework, what we mean by "x is of type τ" is "x is of sort `in` τ".

Though the intentions of the `prop`/`that` p machinery are naturally viewed as different from the intentions of the `type`/`in` τ machinery, the two schemes are isomorphic from the standpoint of our software. Of course it is likely that the additional axioms governing propositions and proofs in a specific theory implemented in our framework will be noticeably different from those governing type labels and typed mathematical objects.

The objects of the sorts `obj`, `prop`, `type`, `that` p and `in` τ are collectively referred to as "entities", and these are in some sense the first class objects of theories built using the framework we are presenting. The objects we have yet to describe are "abstractions" (functions) and while they might in a sense not be viewed as first class objects [this might be a matter of debate] they are certainly essential to getting anything done in this framework.

Briefly, each abstraction is a function of an abstraction sort

$$[(x_1, \sigma_1), \ldots, (x_n : \sigma_n) \Rightarrow \tau],$$

where each x_i is a variable of the entity or abstraction sort σ_i, τ is an entity sort (which may depend on any or all of the variables x_i) and each σ_j may depend on any or all of the x_i's with $i < j$.

If f is of sort $[(x_1, \sigma_1), \ldots, (x_n : \sigma_n) \Rightarrow \tau]$ then $f(t_1, \ldots, t_m)$ is well-formed iff $m = n$ and additional conditions hold which we now present.

1. If $n = 1$ and the sort of t_1 is σ_1, the sort of $f(t_1)$ is $\tau[t_1/x_1]$ (noting the potential dependence of the output sort on the input). If the sort of t_1 is not σ_1, the term is not well-sorted, and is regarded as ill-formed. It is worth noting that the equivalence of the sort of t_1 and σ_1 may be established using definitional expansion or renaming of bound variables.

2. If $n > 1$, the sort of $f(t_1, \ldots, t_n)$ is the same as the output of a function of sort
$$[(x_2, \sigma_2[t_1/x_1]), \ldots, (x_n : \sigma_n[t_1/x_1]) \Rightarrow \tau[t_1/x_1]]^3$$
applied to the argument list $(t_2, \ldots, t_n)$, if the sort of t_1 is σ_1 and the complex term indicated is well-sorted, and otherwise the term $f(t_1, \ldots, t_n)$ is not well-sorted and regarded as ill-formed.

Abstractions may be given as atomic primitives or as complex abstraction terms
$$[(x_1, \sigma_1), \ldots, (x_n : \sigma_n) \Rightarrow D : \tau].$$
If f is defined as
$$[(x_1, \sigma_1), \ldots, (x_n : \sigma_n) \Rightarrow D : \tau],$$

[3]such a function can be presented as the referent of a suitable complex abstraction term: this definition will not fail as the result of no such function being available.

then the sort of f is

$$[(x_1, \sigma_1), \ldots, (x_n : \sigma_n) \Rightarrow \tau],$$

subject to conditions on the term D to be stated:

1. if $n = 1$, the value $f(t_1)$ is $D[t_1/x_1]$ if $n = 1$ if this has sort $\tau[t_1/x_1]$ (otherwise f is ill-formed)
2. if $n > 1$ the value of $f(t_1, \ldots, t_n)$ is the result of applying

   $$[(x_2, \sigma_2[t_1/x_1]), \ldots, (x_n : \sigma_n[t_1/x_1]) \Rightarrow D[t_1, x_1] : \tau[t_1/x_1]]$$

   to the argument list $(t_2, \ldots, t_n)$ (if this is well-formed and has the expected sort: otherwise f is ill-formed).

The definition of substitution requires care in the presence of the bound variables in the abstraction sorts and complex abstraction terms: the difficulties are neatly handled by requiring that all the bound variables x_i in an abstraction sort or complex abstraction term be replaced with fresh variables before any substitution is made into the sort or term (and that abstraction sorts or complex abstraction terms differing only by a renaming of their bound variables are to be identified).

It is further worth noting that in the software and its documentation a complex abstraction term is in effect treated as a subsort (with a single inhabitant) of the sort to which its referent abstraction belongs (so the internally recorded sort of a complex abstraction is itself). This is harmless but should be noted so that one can better understand interactions with the system. Complex abstraction terms also occur as arguments in terms, in which case they do not of course denote their own subsort, but rather its sole inhabitant.

What appears above is a very compact full description of the type/sort system and language of the framework we implement in our software, but this is not the way the user of our software constructs or defines objects. In fact, the user never writes a complex abstraction term (function name or λ-term) in an interaction with the software (at least in its current version) though she may be shown such terms in system responses, nor does the user ever write an abstraction sort. In the user language, all terms standing for abstractions are atomic names, which may appear in applied position or as arguments in application terms $f(t_1, \ldots, t_n)$. It is important to notice that all application terms are of entity sorts.

The user introduces constructions of primitive objects of given sorts (this is how not only primitive notions but also axioms are introduced in this framework) and definitions of objects whose existence follows from the existence of the primitives within the framework of a system of "worlds". The underlying metaphor is that an abstraction is constructed in a given world (the parent world) by postulating or defining an entity of a suitable sort in a further world accessible from that world (which we call the current world) with the entity and its sort depending suitably on entities and abstractions already postulated in the current world (which we may think of as the variable parameters of the abstraction being defined).

The simplest version of the scheme of worlds has the user interacting at any particular moment with a concrete finite sequence of worlds indexed by natural numbers $0, 1, \ldots, i, i+1$. There are always at least two worlds. World i is called the parent world and world $i+1$ is called the current world.

Each world contains a finite collection of declarations of atomic identifiers as being of particular entity and abstraction sorts. The identifiers in each world are given in the order in which they were declared, and the sorts of later identifiers depend only on identifiers appearing earlier in the order on the same world or appearing in some lower-indexed world. All identifiers declared in worlds $0, 1, \ldots, i, i+1$ are available for use in terms and sorts entered by the user in the execution of the basic operations we now describe.

The user may *open* a new world. A new empty world $i+2$ is created and the parameter i is incremented, so the former world $i+1$ becomes the parent world and the new world $i+2$ becomes the current world.

The user may *close* the current world. World $i+1$ is discarded and all of the declarations it contains become unusable. The parameter i is decremented, so the former world i becomes the current world and the former world $i-1$ becomes the parent world. The `close` command cannot be issued if $i = 1$.

The user may *declare* a new entity identifier of a given entity sort. The form of the command is `declare` x σ, where x is a fresh atomic identifier and σ is an entity sort term. The system will check that the atomic identifier declared by the user is fresh and that the entity sort provided for it is well-sorted (a sort `that` p must have the possibly complex term p of sort `prop` and a sort `in` τ must have the possibly complex term τ of sort `type`: there is something nontrivial to be done here). The identifier is declared with the given sort and placed last among the declarations in the current world.

The user may *construct* a new primitive entity or abstraction.

The command `construct` $t : \sigma$ where t is a fresh atomic identifier and σ is an entity sort has the effect of `declare` $t\ \sigma$, except that the identifier t is declared in the parent world (it is a new constant rather than a new variable) and the sort σ may be expanded to eliminate reference to defined identifiers declared in the current world. The sort σ cannot depend essentially on any non-defined identifier declared in the current world.

The command `construct` $f(x_1, \ldots x_n) : \tau$, where f is a fresh identifier, $x_1, \ldots, x_n$ are atomic identifiers declared in exactly that order in the current world (and not defined: these identifiers must have been introduced using the `declare` or `construct` commands), of sorts $\sigma_1, \ldots \sigma_n$ respectively, and τ is an entity sort possibly depending on some or all of the x_i's, declares a new abstraction identifier f in the parent world of sort

$$[(x_1, \sigma_1), \ldots, (x_n : \sigma_n) \Rightarrow \tau],$$

subject to expansion to eliminate all defined identifiers declared in the current world (and this sort cannot depend in an essential way on any non-defined identifier declared in the current world other than the x_i's, which become bound variables in this abstraction sort: the abstraction sort can be thought of as having its own little subworld of the current world in its internal makeup, which contains only the x_i's, with fresh names different from the names appearing in the current world). The restriction that the arguments are given in the order in which they are declared ensures that any dependencies will be of kinds permitted by the type/sort system.

It might appear that we have only allowed variables x_i of entity sorts to be declared, but this is not the case. Constructing an abstraction in the parent world then closing the current world gives an abstraction identifier declared in the current world which can be used as an argument in instances of the `construct` command or the following `define` command.

The user can *define* objects whose existence follows from the existence of the given primitives.

The command `define` $t : D$ introduces the identifier t in the parent world, abbreviating the term D if the latter entity term is well-sorted. The sort recorded for t will be $[D : \tau]$, if τ is the (entity) sort of D: this denotes the subsort of τ with the referent of D (and t) as its sole inhabitant. This is subject to the proviso that any defined identifier declared in the current world which appears in D or in τ is eliminable. D cannot depend essentially on a non-defined identifier declared in the current world. We are defining a constant here in the parent world, and the definition must continue to make sense if the current world is closed.

The command `define` $f(x_1, \ldots x_n) : D$, where f is a fresh identifier, $x_1, \ldots, x_n$ are atomic identifiers declared in exactly that order in the current world (and not defined: these identifiers must have been introduced using the `declare` or `construct` commands), of sorts $\sigma_1, \ldots \sigma_n$ respectively, and D is an entity term possibly depending on some or all of the x_i's , well-sorted with entity sort τ possibly depending on some or all of the x_i's, declares a new abstraction identifier f in the parent world of sort

$$[(x_1, \sigma_1), \ldots, (x_n : \sigma_n) \Rightarrow D : \tau],$$

(this complex abstraction term is interpreted as the subsort of its actual sort

$$[(x_1, \sigma_1), \ldots, (x_n : \sigma_n) \Rightarrow \tau]$$

with its complex abstraction referent as sole inhabitant, and f is understood as referring to that complex abstraction) subject to expansion to eliminate all defined identifiers declared in the current world. The restriction that the arguments are given in the order in which they are declared ensures that any dependencies will be of kinds permitted by the type/sort system.

Note that defined identifiers are only declared in the parent world, but if the current world were closed we would then have defined identifiers in the current world.

Defined identifiers in applied position can be expanded away by substitution (β-reduction, in effect) as indicated above. Defined identifiers declared in the current world will be replaced by complex abstraction terms if they appear in argument position in terms appearing in sorts of objects declared in the parent world: the user never enters complex abstraction terms (indeed, the parser of Lestrade has no provision as yet to handle them), but complex abstraction terms may appear in sorts reported by the system when identifiers referring to them pass out of scope as described here.

The sort checker will view sorts as being the same for purposes of checking terms as well sorted if expansion of definitions (and renaming of bound variables in abstraction sorts) make them the same. The sort checker will only actually expand defined identifiers in the sorts it reports if forced to do so due to the defined identifiers being declared in the current world but appearing in sorts to be recorded in the parent world.

What we have described so far are the core user functions of the software implementing the philosophy of mathematics we are describing. The software is currently called Lestrade (I am Holmes; I hope I may be forgiven for the humor). There are further features of the software, some of which have some additional logical force, but the basic philosophy is implemented at this point, and examples will now be given to illustrate why this can be viewed as an actual implementation of what mathematicians do. It should also be visible at this point that the implementation is a variant of something that already exists: this is a dialect of the ancient proof-checker Automath, though there are instructive differences between Lestrade and Automath. The fact that it is a full implementation of Automath may require some confirmation as well: Lestrade is like the initial variant PAL of Automath in not making explicit complex abstraction terms available to the user, though it in fact gives the user full access to the referents of such terms. The privileges of abstraction sorts are limited in other ways unexpected to an Automath user, which limit the logical strength of the Lestrade logical framework considered by itself, though Lestrade is strong enough to implement any Automath theory (Lestrade will require explicit declarations of primitive constructions by the user to do some things which the primitives of Automath support without any additional user declarations).

We now present evidence that Lestrade implements logic.

```
Lestrade execution:

declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


construct & p q prop
```

```
>> &: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}


declare pp that p

>> pp: that p {move 1}


declare qq that q

>> qq: that q {move 1}


construct Andi p q pp qq :  that p & q

>> Andi: [(p_1:prop),(q_1:prop),(pp_1:that p_1),
>>        (qq_1:that q_1) => (---:that (p_1 &
>>        q_1))]
>>   {move 0}


declare rr that p & q

>> rr: that (p & q) {move 1}


construct Ande1 p q rr that p

>> Ande1: [(p_1:prop),(q_1:prop),(rr_1:that
>>        (p_1 & q_1)) => (---:that p_1)]
>>   {move 0}


construct Ande2 p q rr that q

>> Ande2: [(p_1:prop),(q_1:prop),(rr_1:that
>>        (p_1 & q_1)) => (---:that q_1)]
>>   {move 0}
```

Our first sample of dialogue with Lestrade presents the basic primitives related to conjunction. We introduce variable propositions p and q, and construct the operation of conjunction which constructs a proposition $p \wedge q$ from the propositions p and q.

We then present variables pp, qq and rr of sorts `that` p, `that` q and `that` $p \wedge q$ respectively which we can think of as evidence for the respective propositions. We then declare a rule `Andi` of conjunction introduction which gives a proof of $p \wedge q$ given proofs of p and q, and the two conjunction elimination rules which, given a proof rr of $p \wedge q$ return respectively a proof of p and a proof of q.

```
Lestrade execution:

construct -> p q : prop

>> ->: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}


declare ss that p -> q

>> ss: that (p -> q) {move 1}


construct Mp p q pp ss : that q

>> Mp: [(p_1:prop),(q_1:prop),(pp_1:that p_1),
>>        (ss_1:that (p_1 -> q_1)) => (---:that
>>        q_1)]
>>   {move 0}


open
```

```
     declare pp1 that p

>>      pp1: that p {move 2}


     construct Ded pp1 that q

>>      Ded: [(pp1_1:that p) => (---:that q)]
>>        {move 1}


     close

construct Impi p q Ded : that p -> q

>> Impi: [(p_1:prop),(q_1:prop),(Ded_1:[(pp1_2:
>>            that p_1) => (---:that q_1)])
>>        => (---:that (p_1 -> q_1))]
>>   {move 0}
```

Our second block of dialogue with Lestrade implements primitives related to implication.

The declaration of implication itself is unremarkable.

The declaration of the rule of modus ponens is similar in quality to the rules for conjunction.

The rule `Impi` of implication introduction is more novel. The basic idea is that a function `Ded` which takes evidence for p to evidence for q provides evidence for (a proof of) $p \rightarrow q$. To obtain such a function `Ded` in world 1, we need to open world 2, declare evidence $pp1$ for p, construct evidence `Ded` $pp1$ for q (as a primitive) then close world 2. This gives us a variable `Ded` of the correct type in world 1 and allows us to declare `Impi` in world 0.

It is important to note that the proof of an implication $p \rightarrow q$ is not identified with a function from evidence for p to evidence for q, as in a conventional presentation of the Curry-Howard isomorphism, and as is the case in Automath. For Lestrade, types `that` p are entity types, precluded from actually being inhabited by abstractions (functions). What is done instead is that a construction casts functions from `that` p to `that` q to evidence for $p \rightarrow q$; the rule of modus ponens allows us to present a function correlated with evidence for $p \rightarrow q$ as well. We present a snippet of Lestrade dialogue:

```
Lestrade execution:

%% an experiment in getting a function from proofs
% to proofs from evidence for an implication.

declare ev that p -> q

>> ev: that (p -> q) {move 1}


open

     declare pp1 that p

>>      pp1: that p {move 2}


     define evfn pp1 : Mp p q pp1 ev

>>      evfn: [(pp1_1:that p) => (Mp(p,q,pp1_1,
>>             ev):that q)]
>>        {move 1}


     close
```

We get `evfn`, a function from `that` p to `that` q. correlated with the evidence `ev` for $p \rightarrow q$. We cannot however present a function taking `ev` to `evfn` for technical reasons: a Lestrade abstraction cannot have abstraction output.[4]

[4]We acknowledge that we need to discuss the rationale behind this restriction at some point.

It is a general feature of functions declared so far that they have arguments that seem redundant. In the case of `Impi`, if we are given `Ded` we know what p and q must be, but p and q must appear as arguments. The rule is that the type of `Impi` cannot depend on any non-defined atomic identifier declared in the current world which does not appear in its argument list.

Another interesting point to note is that Lestrade knows about infix notation. A function with two arguments whose first argument is not an abstraction will be displayed as an infix in Lestrade output. Lestrade will read terms $t_1 f t_2 \ldots t_n$ as $f(t_1, \ldots, t_n)$ when t_1 is not an abstraction (and of course when f is an abstraction of the correct sort). Lestrade is not smart about operator precedence: it treats all infix or mixfix operators as having the same precedence and groups to the right (it is never a mistake to use more parentheses, though with one caution to be given below). When an abstraction is followed by its arguments, they may be enclosed in parentheses and may be separated by commas. If the first argument is enclosed in parentheses, the entire argument list must also be enclosed in parentheses. In some cases a comma must be inserted before and/or after an abstraction argument to avoid it being read as a prefix or mixfix operator. Arguments in a mixfix term after the abstraction may be separated by commas, but the entire list of arguments after the abstraction will not be enclosed in parantheses (unless there is only one argument). The Lestrade display shows lots of parentheses and commas.

```
Lestrade execution:

% we try to prove (p & q) -> (q & p)

open

     declare hyp that p & q

>>      hyp: that (p & q) {move 2}


     define step1 hyp:  Ande1 p q hyp

>>      step1: [(hyp_1:that (p & q)) => (Ande1(p,
>>             q,hyp_1):that p)]
>>          {move 1}


     define step2 hyp:  Ande2 p q hyp

>>        step2: [(hyp_1:that (p & q)) => (Ande2(p,
>>               q,hyp_1):that q)]
>>          {move 1}


     define step3 hyp: Andi q p (step2 hyp) (step1 hyp)

>>        step3: [(hyp_1:that (p & q)) => (Andi(q,
>>               p,step2(hyp_1),step1(hyp_1)):that
>>               (q & p))]
>>          {move 1}


     close

define Testthm p q : Impi (p & q, q & p,  step3)

>> Testthm: [(p_1:prop),(q_1:prop) => (Impi((p_1
>>          & q_1),(q_1 & p_1),[(hyp_2:that (p_1
>>               & q_1)) => (Andi(q_1,p_1,Ande2(p_1,
>>               q_1,hyp_2),Ande1(p_1,q_1,hyp_2)):
>>               that (q_1 & p_1))])
>>          :that ((p_1 & q_1) -> (q_1 & p_1)))]
>>     {move 0}
```

Our third block of dialogue with Lestrade is the proof of a theorem, $p \wedge q \rightarrow q \wedge p$. To prove this theorem we need to construct a function taking evidence for $p \wedge q$ to evidence for $q \wedge p$. To define such a function, we open a block in which we declare hypothetical evidence `hyp` for $p \wedge q$, then construct an expression `step3 hyp` which is evidence for $q \wedge p$, whence by abstraction we have constructed a function taking proofs of $p \wedge q$ to proofs of $q \wedge p$ in the parent world, which is suitable to serve as an argument to `Impi` to produce a proof of the desired theorem. Notice that the theorem `Testthm` that we prove is itself a function taking propositions p and q as arguments (so we can construct proofs of all instances of this theorem of propositional logic).

Another point to notice is that we can see expansion of definitions at work. The definition of `step3` includes occurrences of the defined notions `step1` and `step2`, and these identifiers occur in the sort reported for `step3` by Lestrade. But the sort reported for `Testthm` contains no occurrences of the identifiers `step1`, `step2` and `step3`. The occurrences of `step1` and `step2` are in applied position and are eliminated by expansion of definitions. The identifier `step3` occurs as an argument, and is expanded to a complex abstraction term.