Computer theorem proving in math

Carlos Simpson
carlos@math.unice.fr
CNRS, Laboratoire J.A. Dieudonne
Universite de Nice-Sophia Antipolis

Abstract—We give an overview of issues surrounding computer-verified theorem proving in the standard pure-mathematical context. This is based on my talk at the PQR conference (Brussels, June 2003).

Introduction

When I was taking Wilfried Schmid's class on variations of Hodge structure, he came in one day and said "ok, today we're going to calculate the sign of the curvature of the classifying space in the horizontal directions". This is of course the key point in the whole theory: the negative curvature in these directions leads to all sorts of important things such as the distance decreasing property.

So Wilfried started out the calculation, and when he came to the end of the first double-blackboard, he took the answer at the bottom right and recopied it at the upper left, then stood back and said "lets verify what's written down before we erase it". Verification made (and eventually, sign changed) he erased the board and started in anew. Four or five double blackboards later, we got to the answer. It was negative.

Proof is the fundamental concept underlying mathematical research. In the exploratory mode, it is the main tool by which we percieve the mathematical world as it *really is* rather than as we would *like it to be*. Inventively, proof is used for validation of ideas: one uses them to prove something nontrivial, which is valid only if the proof is correct. Other methods of validation exist, for example showing that an idea leads to computational predictions—but they often generate proof obligations too. Unfortunately, the need to prove things is the factor which slows us down the most too.

It has recently become possible, and also necessary, to imagine a full-fledged machine-verification system for mathematical proof. This might radically change many aspects of mathematical research—for better or for worse is a matter of opinion. At such a juncture it is crucial that standard pure mathematicians participate.

The present paper is intended to supply a synthetic contribution to the subject. Most, or probably all, is not anything new (and I take this opportunity to apologize in advance for any additional references which should be included). What is supposed to be useful is the process of identifying a certain collection of problems, and some suggestions for solutions, with a certain philosophy in mind. Our philosophical starting point can be summed up by saying that we would like to formalize, as quickly and easily as possible, the largest amount of standard mathematics, with the ultimate goal of getting to a point where it is concievable to formalize current research mathematics in any of the standard fields. The problems are just those which I have encountered along the way; they are certainly the same as, or similar to, problems that everybody else working in this field has encountered at some time or another, and undoubtedly there is a big degree of overlap in the proposed solutions. Nonetheless, it seems clear that the diverse collection of workers corresponds to a diverse collection of philosophical perspectives, and since the philosophical perspective drives the perception of problems as well as the choice of solutions, it seems like a valid and useful task to set out how things look from a given philosophical point of view. This is a good moment to point out that one should not wish that everybody share the same point of view, or even anything close. On the contrary, it is good to have the widest possible variety of questions and answers, and this is only obtained by starting with the widest possible variety of philosophies.

We can now discuss the difference between what we might call "standard" mathematical practice, and other currents which might be diversely labelled "intuitionist", "constructivist" or "non-standard". In the standard practice, mathematicians feel free to use whatever system of axioms they happen to have learned about, and which they think is currently not known to be contradictory. This can even involve mixing and matching from among several axiom systems, or employing reasoning whose axiomatic basis is not fully clear but which the mathematician feels could be fit into one or another axiomatic system if necessary. This practice must be viewed in light of the role of proof as validation of ideas: for the standard mathematician the ideas in question have little, if anything, to do with logical foundations, and the mathematician seeks proof results for validation—it is clear that any generally accepted framework will be adequate for this task.

There are many motivations for the intuitionist framework, the principal one being deeply philosophical. Another is just concern about consistency—you never know if somebody might come up with an inconsistency in any given axiomatic system (see [27] for example). Common sense suggests that if you take a system with less axioms, there is less chance of it, so it is possible to feel "safer" doing mathematics with not too many axioms. A subtler motivation is the question of knowing what is the minimal axiomatic system under which a given result can be proven, see [44] for example. These questions can even lead to windfalls for "standard" mathematics. In a similar vein, the computer-scientists have some wonderful toys which make it possible directly to transform a constructive proof into a computational algorithm. Thus proof complexity and computational complexity become intertwined.

A growing number of people are interested in doing mathematics within an intuitionist or constructive framework. They tend to be closest to mathematical logic and computer programming, which may be why an inordinate percentage of current computer-proving tools are to some degree explicitly designed with these concerns in mind.

The problem of computer-verified-proof in the standard framework is a distinct major goal. Of course, if you prove something in a constructive framework, you have also proven it in the standard framework but the opposite is not the case. Integrating additional axioms such as replacement or choice into the theory, gives rise to a distinct optimization problem: it is often easier to prove a given thing using the additional axioms, which means that the structure of the theory (i.e. the order in which various things are proved) may be different. The notion of computer verification has already made important contributions in many areas outside of "standard" mathematics, and it seems reasonable to think that it should also provide an important tool in the standard world.

One could also note that, when you get down to the nitty-gritty details, most technical results concerning lambda calculus, including semantics, normalization and consistency results for various (highly intuitionistic) axiomatic systems, are themselves proven in a standard mathematical framework relying on Zermelo-Frankel set theory (see [49] for example). Thus even people who are interested in the intuitionist or constructive side of things might find it useful to have good standard tools at their disposal.

One reason why computer theorem verification in the standard pure mathematical context is not recieving enough attention is that most pure mathematicians are unaware of the subject. This lack of awareness—which I shared a few years ago—is truly colossal, given the literally *thousands of papers* concerning the subject which have appeared in recent years. Thus it seems a bit silly, but one of the goals of the present paper is to try to spread the news.

This ongoing work is being done with a number of other researchers at Nice: A. Hirschowitz, M. Maggesi, L. Chicli, L. Pottier. I would like to thank them for many stimulating interactions. I would like to thank C. Raffalli, creator of the PhoX system, who explained some of its ideas during his visit to Nice last summer—these have been very helpful in understanding how to approach proof documents. Special thanks are extended to the several members of the Coq group at INRIA, Rocqencourt, for their continuing help during visits and on the mailing list. From much further past, I would like to thank M. Larsen for teaching me what little I know about computer programming.

I would like warmly to thank S. Gutt for organizing a great conference, for letting me give a talk which was fairly widely off-subject, and for suggesting a title. And of course the birthday-teenagers for reminding us of the Sumerian numbering system [60].

Basic reasons

The basic goal of mechanized theorem verification is to have a language for creating mathematical documents, such that the reasoning can be checked by computer.

Some of the main reasons why this is of interest to pure mathematicians are as follows.

(1) To reduce the number of mistakes we make in our mathematical work (there are lots of them!).

(2) Dealing with complicated theories—it is difficult to manage development when there are too many things to prove (and often the author doesn't even know exactly what it is he needs to check).

(3) Because of (1) and (2), nobody really has the time or energy adequately to verify all new results, which leads to problems for publication or other timely dissemination. If articles could be validated by machine then referees could concentrate on the question of how interesting they are.

(4) Black-box theories: it is sometimes useful, but currently dangerous, to use results without knowing how they are proven. This can be more subtle than just outright wrongness: an author can have a certain situation in mind, and might use arguments which work in that situation but not in other ones. If the explicit statements of results don't adequately reflect all the hypotheses, then future users had better understand the arguments before blindly applying the statements. An advantage of machine verification is that the language has

to be totally precise. The criterion for inclusion of a result is (and even has to be, see below) that one can directly copy its proof into the new proof development and the verification motor accepts the whole. Thus utilization of a "black box" is via cut-and-paste rather than via importation of a result as an axiom. In this way there is no room for misinterpretation.
(5) Integration of many theories or points of view: it is hard for humans because of the steep learning curve for each new theory.

One can imagine variations on the notion of "black box", where formal proofs are imported in bits and pieces rather than just as results. Thus it might become possible to import *techniques* developped elsewhere, rather than just results; but nonetheless without fully understanding how the techniques work (of course a certain level of understanding would be needed in order to interface things). Even now this is a little-heralded part of mathematical practice, generally glossed over because of the problems of accuracy that are posed. With machine-verified proving these problems should in large part be resolved. In the optimal case, this could lead to a much more diverse fabric in which different workers develop different techniques and import the techniques of others without needing to spend such a long time learning about the details. So, while it might seem at first glance that the notion of "black boxes" and importation of proof techniques would diminish our creativity, actually the opposite may be true when mathematicians will no longer be obliged to spend so much time learning about existing techniques and will have more energy to spend looking for new ones.
(6) New forms of reasoning: now, calculations or proofs with little motivation are ruled out by the fact that the mathematician needs to be motivated in order to understand what is going on. Locally unmotivated reasoning might lead to globally interesting results. A striking example in logic is work on minimal length axioms such as [33] [14], where much of the reasoning was found using theorem-proving software.
(7) The advent of more and more proofs making use of computer calculation (for example the 4-color theorem, more recently a theorem on hyperbolicity of 3-manifolds [16] and the Kepler conjecture [21]) represents *a priori* a somewhat different use of the computer than what we are considering here. However, these proofs pose acutely the problem of verifying logical details, and a formalized framework promises to present a significant improvement. This issue was addressed by B. Werner in a talk at Nice about the 4-color theorem, and has motivated Hales' "flyspeck project" [21].
(8) The possibility of computer-assisted proof verification would significantly

extend the age period in which mathematicians are optimally productive: older mathematicians tend to avert getting bogged down in details, which often prevents full development of the continuing stream of valid ideas.

On the whole, it is a mistake to think that computer theorem-proving will always lag behind regular math. At some point, the advantages will so much more than counterbalance the difficulties that mathematicians using these tools will jump out ahead.

History

Cantor, Hilbert, Russell; then Gödel and so on, worked on providing a strict logical foundation for mathematics.

Church, Rosser, Martin Löf [32], Girard [20] worked more specifically on the logical foundations of type theory.

Automath, the project of de Bruijn, was one of the first explicit projects in this direction; notably he pointed out how to deal with variables (via de Bruijn indices).

In 1977, Jutting entered all of Landau's analysis book in the AUTOMATH system [26].

The first major project involving a large number of people was Mizar; it is still ongoing, and led to the *Journal of Formalized Mathematics* which has been publishing articles verified in the MIZAR system since at least 1989. The mathematical orientation of most of the articles is a little bit far from the mainstream of what we call standard mathematics.

More recently, a number of systems based on some kind of type theory have been developped: HOL, Lego, Isabelle, Nuprl, Nqthm, AC2L, and general "Boyer-Moore" theorem provers...

Coq is a project which seems relatively successful with respect to the considerations discussed below.

An overview of very recent systems would include: Elf, Plastic, Phox, PVS, IMPS. These and many more can be found using an internet search engine with terms such as "Automated theorem proving" or "proof assistant". More generally, a little perseverance turns up an astounding amount of reference material which we couldn't begin to include here (and which is impossible to read in its entirety). A good place to start is with meta-pages such as [66], [67].

Industrial demand, spurred in part by some well-known problems that might have been avoided with use of the appropriate proof technology (Pentium, Ariane 5), continues to be a major impetus for the development of proof engines. In this case, interest is oriented toward proving the correctness of algorithms, integrated circuits or other wiring diagrams, and so forth.

More and more conferences, journals, and mailing lists bear witness to the fact that fields related to computer proof verification are really booming. Browsing the archives of mailing lists such as [68] or [69] is a fascinating homework assignment.

In contrast we might mention that the notion of computer calcuation systems, which is perhaps much better known in the world of pure mathematics, is not quite the same thing as computer proof verification. It is surely related, though, and is also experiencing fast growth.

Some recent examples of proof developments which are of interest to the pure mathematician are: the fundamental theorem of algebra, by Geuvers *et al* [19]; R. O'Connor's formalization of Gödel's theory [37]; the `Reals` and other recent libraries for Coq [52]; the logical libraries in Isabelle [56]; ....

A glance at this history leads to the conclusion that it now seems like all of the ingredients are available so that we could start doing computer theorem verification in a serious way. We just have to find the right combination. It doesn't seem unreasonable to think that this moment will be seen, in retrospect, as an important juncture in the history of mathematics.

Extra-mathematical considerations

Although seeming to step out of the domain of things that we should be worrying about, it is actually useful to elucidate some criteria explaining what would make the difference between a system which would or would not be used and accepted by mathematicians. A good model is mathematical typesetting and TEX. It seems likely that many of the aspects which led to its success would also be good aspects for a proof-verification tool.

Here are a few criteria.

—FREE: the program shouldn't cost anything (and should be free of any other constraints) for scientific users.

—PLAIN TEXT: the main format of the proof document should be a plain text file; this is to insure that nothing is hidden to the user between the document creation and its compilation or verification, and also to insure easy exchange of documents.

—OPEN SOURCE: the basic programming code used to specify the syntax of the language, and to verify the proof document, should be open-source, available to everybody. In the best case (i.e. if the code were simple enough) it could actually be attached as a trailer to the proof document itself, since the program is an integral part of the proof specification (see the discussion of archiving in the introduction of [16]). Open source for this key part of the system brings the guarantee that your document will be readable or compilable at any time in the future.

—REFERENCE MANUAL: an immensely important part of the quality of a theorem-proving system is the quality of its reference manual. This has many aspects, but to be brief, it should be possible to get started using only the reference manual, and at the same time the manual should answer even the most arcane questions (first-time users tend to stumble upon arcane questions more often than one might expect, cf [69]).

—CROSS-PLATFORM: it should be *very easy* to install and use on all platforms. One of the most difficult aspects for a mathematician getting started in this subject is just to get a system up and running usably on whatever computer is available.

—HELLO WORLD: a good test (of the previous two points) would be that the average mathematician, possibly with the help of a somewhat computer-aware colleague but not necessarily the system administrator, should be able to install the system and type out a first text file which is then successfully verified, proving (from scratch) a simple mathematical lemma with a level of difficulty such as uniqueness of the identity element in a monoid. This should take no more than a couple of hours.

—MODIFIABLE: the system should be modifiable by the user, in particular the user should choose (or invent, or re-invent) his own axiomatic framework, his notational system, his proof tactics in a fully programmable language, and so on—these things shouldn't be hard-wired in. In William Gibson lingo, we want something that would give the "Turing police" nightmares. This information would be like the notion of a macro header to a TeX file. People could exchange and copy their header material, but could also modify it or redo it entirely. This is very important for enabling creative research (for example see the subsequent discussion of various points of intuitionist math which could be interesting for standard mathematicians).

—ENCOURAGE MATHEMATICIANS: the people who maintain the system should encourage standard mathematicians to write math in the system; this includes:

—MAINTAINING AN ARCHIVE of everybody's work. People need to be able to start by consulting examples rather than from scratch, and need to be aware of what is already known to avoid duplication of effort; and

—PUBLICIZE THE EXISTENCE OF THE SYSTEM! To be quite specific, this means publicity inside the standard pure mathematical community.
—ATTENTIVENESS TO USER SUGGESTIONS is also an important component of a proof-verification system, particularly in the early phases: pure mathematicians are the scientists who work the most with proof, and it seems logical that their viewpoints would be important to development of a fully functional system. In fact, input from pure mathematicians is likely to be useful even in more far-flung areas of the subject such as industrial applications. This could be seen as compensation for the requirements that the system be free and open-source.

Starting tasks

The hardest part of the subject is the first basic task which needs to be addressed: you need to form an opinion about what you are going to want your documents to look like [46].

This task is of course first and foremost treated by the system designers. However, even within the framework of a given system, the mathematician has to address this task too, in determining the style he will use.

It is subject to a number of constraints and hazards.
—Everything you write down has to have an absolutely well-defined and machine-determinable meaning.
—One must write things in a reasonably economical way, in particular we need a reasonable system of notation for the standard mathematical objects which are going to be manipulated.
—It is better if any difficulties which are encountered, have actual mathematical significance. An example of a difficulty which doesn't seem to have much real mathematical significance arises in type theory when we have two types `A` and `B`, with an element `a:A` and a known (or supposed) equality `A=B`. It often (depending on the type system being used) requires additional work in order to be able to say that `a:B`. To the extent possible, the pure mathematician will want to choose a format which doesn't have this kind of hurdle (or at least in which this hurdle is seamlessly cured).
—At least a part of what is written should be comprehensible, preferably upon simple inspection of the pure text file. This is quite important in view of the

POTENTIAL PROBLEM: it is quite possible (and maybe even generally the case) to invent notation which suggests a meaning different from the actual meaning. When you prove a given statement, did you really prove what you thought you did?

Unfortunately there is no way for the computer to verify this part. The conclusion is that we require human comprehension of the definitions, and of the statements of the theorems (the main ones, at least). This imperious requirement is stronger than the natural desire also to understand the proofs. —Experience suggests that the proofs, in their explicit and verifiable form, actually contain too much information and the reader will not really want completely to understand them. Indeed the origin of the whole project is the fact that we aren't really able to follow the details of our proofs. Instead, there would seem to be two particular aspects of the proofs which should stand out: first, what is the general strategy being used to obtain a proof of a given sort of statement; and second, what unexpected main steps are used in the argument. At the current stage, the theorems whose proofs are attainable are all ones where everybody knows the regular (non-computer) proof, so it isn't currently important that the main mathematical steps stand out. This would change when we get to research-level mathematics.

How it works

We describe in brief some of the salient points of the `Coq` system [52] currently used by the author. A number of other recent systems are similar. Aside from the main—geographical—reason for my choice of system, one can point out that `Coq` scores pretty well (but naturally with room for improvement) on the above lists of *desiderata*. [1]

Some natural considerations lead to the idea of using *type theory* as a basis for mathematical verification [32]. To start with, notice that we want to manipulate various sorts of objects, first and foremost the mathematical statements themselves. Although perhaps not strictly necessary, it is also very important to be able directly to manipulate the mathematical entities appearing in the statements, be they numbers, sets, or other things. To deal with our entities (which will be denoted by expressions entered into the text file), we need function application and the definition of functions. The notation [2] for these are `(f x)` for function application, and

[1] If the makers of other systems feel impelled at this point to say "hey, our system does just as well or better", the main point I am trying to make is: That's great! Try to get us working on it!

[2] To be precise, this is notation from `Coq` version 7.4; in the new version 8 the notation will be different but the reader is referred to the new reference manual for that.

```
[x:-](... x ... x ....)
```

for the definition of the function which, when applied to an arbitrary object say `a`, yields the expression on the right with `a` substituted in place of `x`. This combination of operators yields the *lambda calculus*. The notion of *β-reduction* is that if

```
f :=[x:-](... x ... x ....)
```

then for any expression `y`, the application `(f y)` should reduce to `(... y ... y ....)`. It is natural to ask that the computer take into account this reduction automatically.

The first famous difficulty is that if

```
f:= [x:-](x x)
```

then `(f f)= ([x:-](x x) f)` $\xrightarrow{\beta}$ `(f f)` $\xrightarrow{\beta}$ `(f f)` .... Thus the β-reduction operation can loop.

The solution to this problem is the notion of *type* introduced by Russell. In typed λ-calculus, every "term" (i.e. allowable expression, even a hypothetical or variable one) has a "type". The meta-relation that a term `x` has type `X` is written `x:X`. The λ construction (construction of functions) is abstraction over a given type: we have to write

```
f:= [x:A](... x ... x ....)
```

where `A` is supposed to designate a type. Furthermore the expression on the right is required to be allowable, i.e. well-typed, under the constraint that `x` has type `A`.

THE CHURCH-ROSSER THEOREM states that:
—*In* typed *λ-calculus, β-reduction terminates; also calculation of the typing relation terminates.*

This is a considerable advantage for the end-user: you are sure that the computer will stop and say either *ok* or *no*. And if "no" you were the one who made the mistake.

The product operation Π: in the above expression, what is the type of `f`? We need a new operator Π written as [3]

[3] again this is `Coq 7.4` notation; in `Coq V8` it will be something like
`forall x:A, (B x)`

```
f:(x:A)(B x)
```

where

```
B:=[x:A]"the type of (...x ... x ....) when x has type A"
```

(recall that the allowability restriction on `f` requires that this expression `(B x)` exist, and in fact it is calculable). In the product notation `(x:A)(B x)` the first pair of parentheses is notation, whereas the second pair is just usual parenthetization for grouping together an expression.

Sorts: What is the type of a type? It is clear from the above that we have to start manipulating the types themselves as terms, so we are required to specify the type of a type. This is known as a "sort" (or in some literature, a "kind"). [4] In current theories there are just a few, for example:
`Prop`, the type of propositions or statements, specially if we are using the Curry-Howard principle that a proposition `P:Prop` is considered as a type and its terms `p:P` are the proofs of the proposition `P`; and
`Type_i`, the type of types which are thought of as sets (with a "universe index" `i`, see below), so an `X:Type_i` corresponds to a collection of objects, and an `x:X` corresponds to an element of the collection.

The above system corresponds to Martin-Löf's type theory [32]. One can imagine theories with a more complicated diagram of sorts, which is probably an interesting research topic for computer scientists.

Universes

In order to get a coherent system (i.e. one without any known proofs of `False : Prop`) the sorts `Type_i` have to be distinguished by universe levels `i`, with

```
Type_i : Type_(i+1)
```

whereas `Type_i` $\subset$ `Type_(i+1)` also. The incoherency of the declaration `Type:Type` without universe indices was shown in [20]. Nonetheless, in some type-theoretical programs such as `Coq`, a vestige of `Type:Type` is preserved by a neat, but quirky, mechanism known as *typical ambiguity* (see [12]). With this mechanism, the user doesn't write `Type_0`, `Type_1` and so forth, rather just writing `Type` at each place. During the verification process, the

[4] It's like, kind of a lot of different types of terms for the same sort of thing, you know.

computer makes sure that it is possible to assign indices to all occurences of the word `Type` in such a way that the required inequalities hold. If this is not possible, then a `Universe Inconsistency` error is raised. Thus, the user *writes his text file as if* he were using the inconsistent `Type:Type` system, and the verification program then verifies that no unallowable advantage was taken.

Curiously enough, this leads to a phenomenon that looks for all the world to me like quantum mechanics. [5] Suppose you prove `A -> B` in one text file (where `A` and `B` are some explicit statements and the arrow is to be read as "`A` implies `B`"), and you separately prove `B -> C` in another file. Now one might think that you have proven `A -> C`. However, when you merge the two files together into one big file containing the combined proof of `A-> C`, the verification program might well come up with a `Universe Inconsistency` error on the big file (even if it didn't on the two smaller files). In other words, the constraints on the universe indices which are generated by `A-> B` might be satisfiable, and the constraints on the universe indices generated by `B->C` might also be satisfiable, but the union of both collections of constraints might be unsatisfiable.

This phenomenon can be seen as a "disturbance due to measurement". If we think of `A`, `B` and `C` as measuring posts along a lab bench (such as polarized glass panes, say `A` is horizontal, `B` is at 45 degrees and `C` is vertical) then the result of starting with a photon polarized as `A`, doing the proof of `A-> B` and then "measuring" (i.e. stopping the text file and saying "wow, we just proved `B`"), then inputting the photon polarized as `B` into the next proof of `B -> C`, we get a photon polarized as `C` in the output; whereas if we do the whole proof starting from `A` and trying to get out `C`, no proof-photon comes through!

In [43] (iv), an example of this phenomenon is given using Russell's paradox: we can prove a statement

```
Theorem injections_exist : (X:Type)(term_injection_exists X).
```

which, if re-inputted as an axiom, then allows us to prove `False`:

```
Theorem injections_dont_always_exist :
((X:Type)(term_injection_exists X))-> False.
```

However putting everything together yields a `Universe Inconsistency`. Here

[5] This might be related to [39].

```
term_injection_exists :=
[X:Type](exT ? [f:X->Term](is_injective ? ? f))
```

with

```
Inductive Term : Type := term : (T:Type)T->Term.
```

It is the universe level of the word `Type` in the definition of `Term` which is at issue.

In this file we assume the excluded middle and use `eqT_rect` via Maggesi's proof of `JMeq_eq` [31] [10], but as pointed out by H. Boom [5] it should be possible using [20] to do the same thing without axioms other than the basic `Coq` system.

It is important to note, then, that as long as the typical ambiguity mechanism is in place, it is not allowable to prove a theorem in one file and then import it as an axiom in another file. In other words, it is actually *required* that the proofs of component pieces of the argument be imported by cut-and-paste (this was mentionned above). Luckily the system has a `Require` command that does this automatically without having actually to copy the files.

Questions about "universe polymorphism" lead to some of the most difficult notational problems. The following discussion comes from work of Marco Maggesi, who is also working on solutions to the problem possibly by implementing the thesis of Judicaël Courant [9]. As was pointed out in [12], the basic example is the distinction between small categories and big categories. This is fundamentally a question of assigning universe levels in the definition of the notion of category. Ideally, polymorphism should be extended to definitions such as this. In the absence of such a mechanism, we seem to have to rewrite definitions such as `Category_i`, once for each different universe level `i`. This can lead to a nightmare of notation, for example do we need to distinguish between a `comp_sm` for composition in a small category, and `comp_bg` for a big category (and the same for all the other functions which enter into the definition)? If this seems annoying but doable, notice that we then need notions of `Functor_ij` for functors from a `Category_i` to a `Category_j`; and composing them, and so forth!

One of my main motivations for going back to a purely set-theoretic approach is to try to avoid the above problems as much as possible. However, it is impossible to avoid them completely: it seems that there will always be a threshold between the objects we are willing to manipulate and the bigger ones we would prefer not to manipulate, and looking at categories of the smaller objects inevitably bumps up against the bigger ones. Until someone comes up with an adequate general solution (perhaps the ideas in [12] are relevant) we are left with little choice but to live with this situation.

Inductive objects

An interesting feature of a number of systems including `Coq` is the notion of inductive objects, which allows us automatically to create recursive datatypes. Since understanding and calculating with recursive datatypes is what the computer does best, it can be good to make use of this feature. It is often the only viable way of getting the computer hitched onto the calculational aspects of a problem. (This might change, notably with the possible advent of connectivity between theorem proving systems and computer calculation systems.)

We won't go into any further detail on the basic structure or functionality of inductive definitions; the reader is referred to the reference manual of [52].

Exotic versus regular math

The system described above applies to a wide variety of situations, problems and logics.

INTUITIONISM: One of the main utilizations of this type of system is to verify intuitionistic logic. In fact it is difficult to imagine taking a complicated argument and trying to insure, as a human reader, that the proof *didn't* use the excluded middle in some hidden place in the middle. In this sense, computer verification seems crucially important and allows a much more serious approach to the field.

IMPREDICATIVE SORT: in `Coq` for example, a third sort `Set` is added. It represents constructive setlike objects. The main property which sets it apart from say `Type_0` is that it is *impredicative*, in other words if `X:Set` then the product type

```
Set -> X := (Y:Set)X
```

is classified as having type `Set`. We have simplified here but `X` could also depend on `Y`. Nonetheless elements `X:Set` behave somewhat like sets (in technical terminology, strong elimination is allowed, so that constructors in inductive sets are distinct). The intuition behind this situation is that the constructive functions on a constructive set themselves form a constructive set too. Thus these considerations are closely related to the constructive or algorithmic aspects of mathematics. Hugo Herbelin pointed out to me that this can lead to some unexpected and fascinating things, such as a proof which extracts to an algorithm where nobody can understand how it works [23].

Caution: the existence of an impredicative but strongly eliminatory sort is somewhat "orthogonal" to standard mathematics, in that it contradicts most choice axioms you can imagine, even the much weaker dependent choice axioms [17] [7].

Perhaps not all, though: David Nowak in [36] proposed doing the axiom of choice for the `Ensembles` library, which means looking at a function associating to any predicate `P:X->Prop` another predicate `(choice P):X->Prop` with the property that if `P` is nonempty then `(choice P)` is a singleton. One conjectures that this version is consistent even with an impredicative `Set`.

Nowak's version is difficult to use in practice, and leads to a situation where the only objects which are manipulated are predicates. This nullifies much of the advantage of using a type-theory framework. So, leaving it aside we can say that the axiomatic setup we would like to use for standard mathematics is contradictory in the full version of `Coq` with its impredicative `Set`. This is a problem which will apparently be addressed in the new Version 8 of the system, by incorporating a switch allowing us to turn off the impredicativity of `Set`. In the meantime, we resolve informally by convening never to use the impredicativity, when we work with our standard axioms.

Logic in a topos: recall from Moerdijk-MacLane [35] that one way of providing a model for certain intuitionistic logical systems is by looking at logic in a topos other than the topos of sets. One can say this differently (as they do in the book) by saying that any logical reasoning which is expressed solely in a limited intuitionist formulation, applies internally to logic within any topos. One can formulate as a conjecture, that this principle extends to the full intuitionist logical system as implemented in `Coq`:

Conjecture *The $\lambda - \Pi$ calculus with universes and inductive objects as described above and implemented in* `Coq`*, applies to logic within a Grothendieck topos.*

It would be a generalization of the principles set out in [35]. Some part of it may already be known, see Streicher [45] as well as the references therein (and indeed the formulation of the question appears to have been known to workers in Synthetic Domain Theory for some time [13] [42]). It is not immediate, for example, to see how to implement the role of the sorts `Type_i`. A *caveat* might also be in order because recent versions of `Coq` include some inductive principles such as `False_rect` or `eqT_rect` which might not be realizable in the internal logic of a topos. So, the above statement is really just intended to point out that it is an interesting research problem to see how far we can extend the internal-logic principle from [35].

As an example of how this type of thing might actually lead to useful results for standard mathematics, one can use the full flexibility of the `Coq` implementation of type theory to define internal cohomology within the logic (this is done for H^1 in [43] (v)). If this could be applied to a Grothendieck topos, it would show that cohomology in the topos was an internal logical construction. Internal algebraic geometry inside the topos of sheaves on the big fpqc site takes on a particularly nice form very close to the original Italian geometry, for example affine n-space is quite literally just the cartesian power of the ground field k^n.

THE HODGE CONJECTURE: This type of thinking might be related to it. The statement of the Hodge conjecture implies that the problem of saying whether a given topological cycle is algebraic, is decidable. Same for the question of whether a given cycle is Hodge (i.e. vanishing of the Hodge integrals). This might be considered as a logical analogue of Deligne's observation that the Hodge conjecture implies that Hodge cycles are absolute. The pertinance of calculatory methods showed up recently in [41]. It could be interesting to investigate further, for example does Deligne's proof of [11] imply the decidability for abelian varieties? And in exactly which level of intuitionist logic would the decidability hold? What are the consequences of this decidability given already by the Lefschetz $(1,1)$-theorem? What happens internally in a topos? Could one obtain a counterexample to the Hodge conjecture by showing that in some cases the question is undecideable?

So there are undoubtedly lots of fun things to look at in these exotic reaches, but they don't really have anything to do with our original goal of implementing, as quickly as possible, a large amount of standard mathematics. More generally, my experience is that any "brilliant gadget" seems predestined to be a bad idea. Boring as it may seem, we have just to plod along in a mundane but systematic way.

Typed versus untyped style

One of the central questions in the theory has (for a long time) been whether to impose a strict type system, or whether to relax the typing constraints. The present paragraph is sort of floating in the middle, because on the one hand our description of the `Coq` system given above presupposes at least some level of acceptance of their point of view that type theory is the way to go; on the other hand the discussion previews our choice of options in the next paragraph.

There are some very interesting threads of discussion about the issue in the mailing lists, particularly [68] following the postings [28] and [24]. It is interesting to note that in the first thread of discussion, Lamport (forwarded by Rudnicki [28]) offered the viewpoint that the untyped style was better than the typed one. He was promptly refuted by almost all the subsequent messages. A year later when Holmes [24] asked what options people generally used to answer the question, almost everybody replied that the untyped style was obviously the way to go. Unfortunately, this seems to have marked the end of interesting discussion on the QED mailing list [68]. At about the same time the Coq system (where typed language was systematic) started picking up speed: the announcement of Coq Version 5.10 is among the last group of messages in Volume 3 of [68].

The published version of [28] is [29], issued from a combination of Lamport's original note and a rebuttal by L. Paulson. This discussion and the discussion in J. Harrison [22] are excellent references for the multiple facets of the question which we don't attempt further to reproduce here.

The distinction between the typed and the untyped style is generally seen as related to the question of whether or not one assumes the axiom of choice. In the absence of the axiom of choice (in particular, in any sort of constructive environment) it is much more natural to keep a lot of type information, since the idea there is to look at the values of functions only when they are well defined and make sense. On the other hand, the axiom of choice provides a nice way to implement aspects characteristic of an untyped environment such as function evaluation which is defined everywhere (see [29], this is also recalled briefly below).

Some authors make a distinction between choice and dependent choice, the latter stating only that we can choose a function whose values lie in a family of single-element types. By [17] [7] this weaker form poses the same problems vis-a-vis a constructive axiom such as impredicativity of `Set`.

Furthermore it generates a lot more proof obligations since one has to prove unicity whenever it is used. For this reason we don't pay too much attention to the distinction, and when thinking of assuming choice we look directly at the full axiom, even a strong Hilbertian form involving an ϵ-function.

Options for doing set theory

Sets are the most basic building blocks entering into the definition of various mathematical objects. On the other hand, as we have seen above, natural considerations about general mathematical theories lead to type theory as the framework for the system as a whole. The most critical decision about document style is therefore the way the notion of a set is encoded in the ambient type theory.

We will review some of the most prominent options. These follow the contours of the "typed versus untyped" discussion.

1: Interpret sets as types. Benjamin Werner's paper providing the justification for the $\lambda - \Pi$ calculus with inductive definitions [47] rests on a *semantics*, i.e. assignation of meaning to the formal symbols of the theory, where `X:Type_i` corresponds to a set in the ith Grothendieck universe U_i, and functions such as constructed by the λ operation `[x:A]( ... x ... )` are represented by sets which are the graphs of the corresponding set-theoretic function. We can use this semantics as our method for interpreting sets, namely when we want to talk about a set we just take any `X:Type`. In this option we make full use of the notion of dependent types, see C. McBride [3]. The following example shows how we would define the type of groups. It uses the `Record` construction which is a special inductive definition corresponding to an explicit list (of objects possibly depending on the previous ones).

```
Record Group : Type_(i+1) := {
elt:Type_i;
op : elt -> elt -> elt;
id : elt;
inv : elt -> elt;
assoc : (x,y,z:elt)(op (op x y) z) = (op x (op y z));
left_id : (x:elt)(op id x) = x;
right_id : (x:elt)(op x id) = x;
left_inv : (x:elt)(op (inv x) x) = id;
right_inv : (x:elt)(op x (inv x)) = id
}.
```

With this definition, if `G:Group` then `(elt G):Type_i` is the "set" of elements of `G`.

The advantage of this viewpoint is that it is generally quite easy to understand what is going on. You the uninitiated reader didn't have any trouble reading the previous definition, and could by cut and paste give the analogous definition of, say, `Ring`, right?

One of the drawbacks with this point of view is that if `H:Group` too, and if for some reason we know `G=H`, we have problems interpreting elements of `G` as being the same as elements of `H`. For example it is not legal to write

```
(x,y,z:(elt G))(op H (op H x y) z) = (op G x (op G y z))
```

(of course one would never write such a thing outright, but this sort of thing tends to come up in the middle of proofs). This problem is alleviated by the recent inclusion in `Coq` of the "transport" function `eqT_rect`, the utilisation of which is best accompanied by a systematic exploitation of the `JMeq` equality proposed by McBride [3]. This doesn't make the problem go away entirely, though.

This point of view adheres pretty closely to the "typed" side of the discussion in the previous section. It isn't written in stone, though. An appropriate choice axiom can allow us to extend function definitions to larger types, whenever a default is available. Unfortunately this implies a distinction between empty and nonempty types. Some have made more exotic proposals to include a default element $\perp$ in every type (this is based on work of Scott [40], and was mentionned in the replies to [24]). I haven't looked closely at it, mostly for fear of a large number of proof obligations of the form $x \neq \perp$, and because it obviates the advantage that we can easily understand what is going on. Also because the guy in the office next door talks about it so much.

To close this subsection we also mention another more subtle drawback, which might really be the main problem. There is a very slight shift in universe indexing, compared with what we are used to in set theory. Notice in the above `Record` that if we want the underlying set `(elt g)` to be in the universe `Type_i`, the type `Group` is forced to be in the universe `Type_(i+1)`. In general, a set at a given universe level will also have elements at that level. Thus, *a priori*, the elements `g:Group` are to be considered (even though this statement doesn't have any precise technical meaning) as being in the $(i+1)$-st level. This contrasts with set theory where: if `g` is a group whose underlying set (`elt g`) is in the i-th universe, then `g` is represented by an ordered uple which is also in the i-th universe.

2. Setoids. From a constructive point of view, the notion of a quotient can be problematic because an equivalence relation need not be decidable, so when defining an object as a carrier modulo a relation, the carrier might be constructive or computational but the quotient less so. Another problem is that the classical construction of a quotient as a set of equivalence classes can pose problems in certain intuitionist frameworks (imagine doing that internally in a topos, for example).

For these (or perhaps other) reasons a much-favored implementation of sets is as *setoids* (cf the `Setoid` libraries in `Coq`). A setoid is a carrier type together with an equivalence relation. We can think of them as quotient sets without adding additional axioms.

One major drawback is that it generates a lot of extra stuff to prove, basically that all operations defined on the level of the carrier type are compatible with the equivalence relation.

For standard mathematics, we don't really need the setoid viewpoint since we are willing to add in all of the axiomatic baggage which makes the standard theory of quotients go through.

What is useful is to know that such structures exist, are natural, and have been studied, because they occasionally show up uninvited.

3: Introduce a type `E` of sets. In this viewpoint, the type-level is considered as the meta-language of mathematical expressions, and we introduce maybe some parameters [6]

```
Parameter E : Type.
Parameter inc : E -> E -> Prop.
```

Here `E` is for "Ensemble" and `inc` is the elementhood relation (we read `(inc a b)` as saying "`a` included—as an element—in `b`"). One can then go on to suppose all of the usual axioms in set theory, and proceed from there. For example the definition of the subset relation becomes

```
Definition sub := [a,b:E](c:E)(inc c a) -> (inc c b).
```

A development of set theory using this type of viewpoint was done by Guillaume Alexandre in his Coq contribution "An axiomatisation of intuitionistic Zermelo-Fraenkel set theory" (1995) [2]. This version is also quite

[6] In `Coq` terminology a parameter is something which is axiomatically assumed to exist and to be given.

analogous to the foundations of the MIZAR system [59], and has been developped rather deeply in the "ZF" logic in the Isabelle system [56].

One can even *create* such an `E` using an inductive definition and a quotient, see Aczel [1] and Werner [47]:

```
Inductive Ens : Type := sup : (A:Type)(A->Ens)->Ens.
```

This is a carrier type, bigger than what we actually think of as *Ens* because an equivalence relation of extensional equality is defined (thus, `E` should actually be thought of as a setoid in Werner's development, and to get a type we would have to take the quotient). Once we have the basic axioms, which in these constructions may require supposing their counterpart axioms on the type-theory side [47], the further development of the theory doesn't require knowing about the construction, so in this sense it is conceptually easier just to take `E` as a parameter.

In theories of this option, sets are terms of type `E`, but they are not themselves types. The relation of elementhood is just a propositional relation on the type `E`. This is contrasted with the type-theoretical interpretation (1) where the relation `x:X` is meta to the theory (not actually accessible from within) and is decided (and decidable) by the computer.

This would seem at first glance to be much less powerful than version (1) where sets are types. In a certain sense that's true, but in some other ways it is actually more powerful. The fact that the elementhood relation is a propositional function in the theory allows us to subject it to mathematical manipulation. In particular, the problem referred to above about elements of sets which are equal, goes away: if `a=b` and if `x:E` then the propositions `(inc x a)` and `(inc x b)` are equal since they are functions of `a=b`. Inside proofs, the fact that a certain term has a certain type is replaced by the statement that a certain set is an element of another set; in the first case the knowledge is required at the start because otherwise the very statement that is being written will be rejected, but in the second case the proof can be deferred into a separate branch of the proof tree.

If we further add an appropriately strong version of the axiom of choice (basically Hilbert's ϵ) then we get an even better situation [28] [29] where function evaluation can be defined between any two sets:

```
Definition Function.ev :=
[f,x:E](choose [y:E](inc (pair x y) f)).
```

In natural language this says that `Function.ev` is defined as the function which to any `f,x:E` assigns a choice of `y:E` such that the pair `(pair x y)` is contained in `f`, if such a `y` exists. Here `choose` is the Hilbert ϵ function which chooses something if it exists, and chooses anything else if the thing didn't exist. In fact, if we write down all of the axioms in this general spirit of everywhere-definedness, then everything that follows will be everywhere-defined too. This leads to a considerable simplification because we are no longer bothered with the multiple problems caused by dependent types.

The main disadvantage of the purely set-theoretical approach is that we have lost any possibility for the computer directly to compute on sets, because computation requires specific knowledge about the inductive structure of the type containing a given term and our `E` no longer has any inductive structure. (This might well be alleviated using the direct constructions of [1] and [47], though, a thought-experiment that starts off the next option.)

4: Combine 1 and 3. In the Aczel-Werner type constructions (for example [47]) one can construct a set i.e. term `x:E` whenever one has a type `A:Type` together with a realization function for its elements `A -> E`. Imagine that we want to realize as a set an inductive type say

```
Inductive nat : Type :=  O : nat | S : nat -> nat.
```

To do this we would need to construct the realization function on elements, i.e. an injection `nat -> E`. In this particular case it is natural to do so because the construction

$$\emptyset, \ \ \{\emptyset\}, \ \ \{\emptyset, \{\emptyset\}\}, \ldots$$

played an important historical role in the beginning. However, if we try to do the same for, say, a free monoid, few people ever really worry about the details of how to do it. This requirement would therefore seem to qualify as something that doesn't have any real mathematical significance: we only care about the universal properties of an inductive type, not about how it is actually realized.

In Werner's paper [47], he describes (albeit in a rather brief way) how to realize inductive types as sets, and consequently how to realize every type as a set. So, let's just abstract out this process and suppose, with a parameter, that we know how to do it but we don't care what it is. We get a situation where, combining (1) and (3), sets are types, but also the elements of sets are realized as sets themselves—which after all is the bedrock assumption of classical set theory.

This is the viewpoint which is developped in the first three files in [43]. We start by putting

```
Definition E:= Type.
Parameter R : (x:E)x->E.
Definition inc := [a,x:E](exists [b:x](R b) = a).
```

Thus, we identify the notion of set with the notion of type. The parameter `R` is the realization function which says that the elements of a set/type are themselves sets. The elementhood relation `inc` is no longer a parameter: it is defined using `R` by saying that $a \in x$ iff there exists a term b of type x whose realization is a.

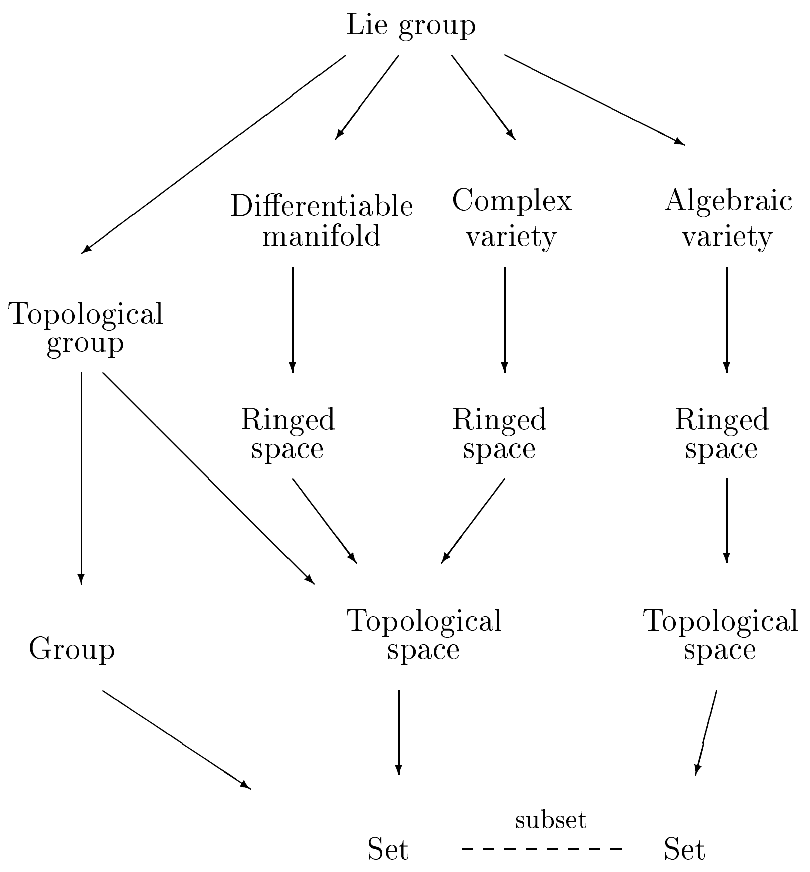
We assume a certain number of axioms such as choice (in the form of a Hilber ϵ-function) and replacement. The axioms also contain a few things designed to specify that `R` is the function we said it was. They are listed further on below.

We can then go on to develop set theory for `E` just as would be done in option (3) above. On the other hand, we can use some inductive types and other type-theory features. Notably these are used to establish a notational system, as will be explained in further detail below.

Inheritance

We indicate here a sample problem. One encounters many many problems and this is only one of them.

Consider the notion of Lie group. This fits into at least 4 distinct (plus one overlapping) inheritance threads as follows:



The three rightmost threads are formally similar, however the ringed spaces in question are different, and the topological spaces for the two middle threads are the same but different from the (Zariski) topological space in the "algebraic variety" thread. One could identify other inheritance strings, and note that there are intricate relationships between them, for example the structure sheaf of the ringed space underlying the complex variety is a subsheaf of the structure sheaf of the ringed space underlying the differentiable manifold, and the set underlying the usual topological space in the first threads is the subset of closed points in the set underlying the Zariski topological space.

If G is a Lie group, then we tend to use the letter "G" for any and all of the above things and more.

On the other hand, the details of these inheritance relations would have precisely to be spelled out in any computer verification system.

Letting the same letter "G" mean any of an amorphous mass of notions, represents a huge collection of notational shortcuts. While clearly impossible to keep all of them, it should be possible to keep some. This could lead to some difficult choices: how to decide which parts to keep and which parts to relegate to a more complicated notation. And in any case, how do we minimize the amount of additional notation necessary to make these shortcuts precise?

The example of a Lie group is deliberately complex in order to illustrate the problem. However, it comes up to varying degrees in pretty much all of modern mathematical practice. We don't have a good solution, however it is hoped that the notational system started in [43], using option (4) of the previous section and described a bit more below, might be of some help.

The attached development

The development attached to the source file of this preprint [43] might provide an example of what type of effort should be necessary in order to find the right combination of ingredients. We are by no means claiming that it is the right way to go, on the contrary people should strongly be encouraged to try out, as I have and will continue to do in the future, many different combinations of axioms, semantics and styles. In order to advance the subject, it is important that each of us let everybody else know about these attempts.

For information, here are the parameters and axioms at the start of the file ([43] (i)):

```
Definition E := Type.

(****** Elements of sets are themselves sets ******)
Parameter R : (x:E)x -> E.
Axiom R_inj : (x:E; a,b:x)(R a) == (R b) -> a == b.

Definition inc := [x,y:E](exT ? [a:y](R a) == x).
Definition sub := [a,b:E](x:E)(inc x a) -> (inc x b).

(***** Extensionality *******)
Axiom extensionality : (a,b:E)(sub a b) -> (sub b a) -> a == b.

Axiom prod_extensionality : (x:Type; y: x -> Type; u,v: (a:x)(y a))
((a:x)(u a) == (v a)) -> u == v.
Lemma arrow_extensionality : (x,y:Type; u,v:x->y)((a:x)(u a) == (v a)) -> u == v.
Intros x y. Change (u,v:(a:x)([i:x]y a))((a:x)(u a)==(v a))->u==v.
Intros. Apply prod_extensionality. Assumption. Save.

(***** Choice *********)
Inductive nonemptyT [t:Type] : Prop :=  nonemptyT_intro : t -> (nonemptyT t).
```

```
Parameter chooseT :(t:Type; p: t-> Prop)(nonemptyT t) -> t.
Axiom chooseT_pr : (t:Type; p: t-> Prop; ne: (nonemptyT t)) (exT ? p)-> (p (chooseT p ne)).

(***** Replacement ******)
Parameter IM : (x:E)(x -> E) -> E.
Axiom IM_exists : (x:E; f:x -> E; y:E)(inc y (IM f)) -> (exT ? [a:x](f a) == y).
Axiom IM_inc : (x:E; f:x -> E; y:E)(exT ? [a:x](f a) == y) -> (inc y (IM f)).

(***** these follow from Choice but are included for brevity **************)
Axiom excluded_middle : (P:Prop)(not (not P)) -> P.
Axiom proof_irrelevance : (P:Prop; q,p:P)p==q.

(***** Identify equivalent propositions ***************)
Axiom iff_eq : (P,Q:Prop)(P -> Q) -> (Q -> P) -> P==Q.

(*** Historically nat is realized as the set of standard finite ordinals ***)
Axiom nat_realization_O : (x:E)~(inc x (R O)).
Axiom nat_realization_S : (n:nat; x:E) (inc x (R (S n))) == ((inc x (R n)) \/ (x == (R n))).
```

One of the main questions which I have tried to address is that of notation. In Bourbaki and thereafter, phrases of the form *"un ensemble muni de . . . "* occur profusively. To machine-interpret them, one needs a precise definition of the verb *"munir de"*. To give an example, consider the operation "plus". Many objects contain an operation which we habitually denote by "plus", for example rings, algebras, modules, abelian groups, not to speak of the topological (or metrized) versions of all these things. In a purely type-theoretical approach, one must specify different notation for the different "plus" operations for all these objects. This is actually fairly common and can lead to an excess of non-intuitive notation. On the other hand, in a classical set-theoretic approach, one has to take care about the order in which the operations are put into an n-tuple, for example a ring could be a triple with the underlying set, the plus function, and the times function. If (by some strange accident) we think of a module as a triple with first the underlying set, then the scalar multiplication function, then the plus function, then the extraction function which yields "plus" would be different in the two cases (pr_2 versus pr_3). Of course it is unlikely that one would make such a choice: more natural would be to have "plus" occupy the second place in both cases. However, when we include the notion of algebra, things become more complicated: whereas a ring has an operation "times", and a module has an operation "mult" (for scalar multiplication), an algebra has both of these operations (and they might be distinct functions, for example there is nothing saying that the ring of scalars has to be a subring of the algebra, so the "mult" function is not subsumed by "times"). In the point of view where objects are simple n-tuples, the question then arises as to whether the third space should be occupied by the "times" or the "mult" operation: there is a conflict between maintaining the same notation for "times" across rings and algebras, and maintaining the same notation for "mult" across modules and algebras.

The reader will probably find that, when stated this way, there is a relatively obvious solution, and this is the one which we have implemented. It dates back at least to the discussion of record types as functions on strings in [29].

Rather than being n-tuples, objects should be thought of as functions where the domain is a subset of a fixed set of "tags" denoting the various types of operations which we want to have. Thus there would be a tag `Underlying` for the underlying-set function; a tag `Plus` for the plus function; a tag `Times` for the times function; and a tag `Mult` for the mult function. Respectively, rings, modules and algebras are implemented as functions whose domains are

$$Dom_{Ring} := \{\texttt{Underlying}, \texttt{Plus}, \texttt{Times}\},$$

$$Dom_{Module} := \{\texttt{Underlying}, \texttt{Plus}, \texttt{Mult}\},$$

and

$$Dom_{Algebra} := \{\texttt{Underlying}, \texttt{Plus}, \texttt{Times}, \texttt{Mult}\}.$$

In this way, the various operations are obtained by simply evaluating at the corresponding tags, so the "mult" functions for modules and algebras are identical and can be assigned a single name. As a first approximation we can write:

```
mult :=
[a,x,y:E](Function.ev (Function.ev (Function.ev a Mult) x) y).
```

In the actual file we take a somewhat more abstract approach and give a general encoding for multivariable functions (inspired by Capretta [6]) which formalizes the sequence of three occurences of `Function.ev` in the above, but for commodity the end result is a slightly different function (this doesn't really matter though). An explanation of that mechanism would go beyond the scope of the present note and wouldn't be very interesting or useful, so the reader is referred directly to the source file. Perhaps the only interesting point to note is that we make use of the inductive structure of the datatype `nat` in order to define `n`-variable functions for any `n:nat`. This is an example of the benefit obtained by mixing type theory and set theory.

Note here that the lack of type constraints and the axiom of choice come into play, because they allow us to define function evaluation on any element as discussed previously.

A systematic application of the above principle allows a considerable amount of notational simplification. It also allows another kind of simplification: all objects will use the same tag `Underlying` for their underlying-set functions. This allows us to develop the theory of morphisms between underlying sets, independantly once and for all. This is done in the module `Umorphism`. The function assigning to an object `a` its underlying set is denoted `a\mapsto (U a)`, with `(U a)` being technically a choice of evaluation of `a` on the element `Underlying`, thus:

```
U:= [a:E](Function.ev a Underlying).
```

A Umorphism is a triple, or rather an object whose domain consists of three tags `Source`, `Target` and `Mapping`. The axioms on such a triple `f` are that `(mapping f)` is a function whose domain is `(U (source f))` and whose range is contained in `(U (target f))`, where `U` (resp. `source`, `target`) denotes the function of evaluation on the tag `Underlying` (resp. `Source`, `Target`). We can define composition, identities, inverses (when they exist), inclusions, and various lemmas about these operations, for this general notion. Which all then applies directly to the theory of morphisms for objects whenever the morphisms can be faithfully expressed as maps between the underlying sets (which is really very often the case). Morphisms between such objects are Umorphisms which are subject to additional conditions, but the constructions such as composition and so forth are the same as in general.

We now come to another place where a mixture of type theory (with inductive definitions) and set theory can help. In the above discussion, we rather rapidly glossed over the question of what the "tags" actually are in set-theoretical terms. This is not totally anodine, because in any given object, the tags used for different pieces of the structure have to be distinct. The first and most obvious solution is to assign differing integer numbers to the different tags; this would lead to a sort of "Dewey decimal system", where we might assign for example 20-29 to category theory, 30-39 to algebra, 40-49 to topology and so forth. This unfortunately leads to a quadratic collection of proof obligations: if we have n different tags in play, then we need the $n(n-1)/2$ statements that they are pairwise distinct. This turns out to be a lot of different silly things to prove. A big improvement is obtained if we rely on `Coq`'s inductive types: we can for example just define an inductive type containing all of the different tags as constructors. The statement that the constructors are different is the "Discriminate" tactic (see also [34]). In fact we don't even need to rely on this tactic because the difference between the constructors is integrated automatically into the inductive proof of the statements where we need it, basically the statement that if you construct an object and then destruct it you get back what you put in.

The only problem with the approach outlined up until now was that it required putting in all of the tags which one wants to use, at the point where the notational inductive type is defined. This results in an environment like "C" where you have to declare your data names at the beginning of the development.

A last improvement, getting rid of this problem, is to define the notational inductive type (basically a string but also with the arity included) `nota` with 27 constructors, first one for each letter of the alphabet (26 constructors of type `nota -> nota`) and a last constructor denoted `dot : nat -> nota`. Then we literally spell out the elements of `nota` that we want to use, with the `dot` notation giving the arity of the operation. For example

```
Underlying := (U_ (N_ (D_ (R_ (L_ (dot (0))))))).
```

This specifies `Underlying` as a tag (i.e. an element of `nota`) corresponding to an operation of arity 0. Similarly,

```
Plus := (P_ (L_ (U_ (S_ (dot (2)))))).
```

Here the arity is 2. Note that it is a good idea to use rather short abbreviations because otherwise the Cases constructions which distinguish between different notations are expanded into huge monsters. With strings, anybody can add new tags (or rather, the pool of tags is free—with one generator for each arity—over the infinite semigroup on 26 generators, so there is always room to look at new tags).

With this notational system we make a first stab, in the file [43] (iii), at treating some standard mathematical objects. In this case, ordered sets. At the end we obtain a proof of Zorn's lemma. It should again be stressed that this is not new, for example it is done in Isabelle [56]. Rather the point is to try out our notational style to see if it is reasonable. My conclusion was that there is probably lots of room for improvement, but that one can at least imagine getting to more modern stuff using the system.

While this effort certainly doesn't represent the ultimate optimal solution to the problem of fixing adequate notation, it nonetheless should serve to indicate where the main problems are, and at least indicate that there probably exist solutions. In this sense it should increase our confidence that the stumbling block of notation isn't an impossible hurdle, and that it should be possible to go on to other things.

Perspectives

We now outline a sketch of what some part of the further development of machine-verified mathematics might look like. This could start with [43]; or any of a number of other basic supports such as [56], [59], [52], ..., and indeed some of these are already quite far along.

Most objects are constructed with a single common variable, denoted `(U a)`, thought of as the "underlying set" of `a`. Most types of mathematical objects have underlying sets, and furthermore the morphisms between objects can be considered as maps between underlying sets, satisfying additional properties. This makes it possible to develop a certain amount of categorical machinery (e.g. composition, identities, inverses) at once for all these types of objects.

With this in mind, we can define monoids, groups, rings, algebras, modules; categories, eventually with Grothendieck topologies; presheaves (and sheaves if the category has a g.t.); ordered sets; topological spaces also eventually with other structures so that one could define topological groups and so on.

The main work which needs to be done next is everything which has to do with counting: the notion of ordinal, the notion of equivalence of cardinals, and the smallest ordinal of a given cardinality; the notion of finiteness, and various facts about finite sets (notably that any automorphism of a finite set decomposes as a product of transpositions); various notions of collection or list of objects; etc. The theory of transpositions should help for defining multi-term summation in an abelian monoid.

Next, the counting system has to be extended to the numbering system. We can import much of the beautiful development of Peano arithmetic already in the `Coq` system, as well as the `ZArith` library containing a logarithmically fast implementation of integers. Then we need to define the rationals as a localization of the integers (and for this we might as well call upon the general notion of localization of a ring). Then we can define the reals as the set of Dedekind cuts of the rationals. It is not clear whether it is better to import the real number results from the existing `Coq` library, or to redo a substantial portion, since our approach using things like the axiom of choice, and an untyped point of view, might give substantial simplification here (as opposed to the case of elementary arithmetic where we wouldn't expect any such benefit).

Once we have the reals (and therefore the complex numbers too) we need to establish the basic facts about the topology on these spaces, and along the way note that this opens up the definition of metric space, so we can do the basic facts about metric spaces. This could eventually lead to stuff for analysis such as Hilbert or Banach spaces, and all of the basic theorems of analysis. This is not necessarily urgent although it would be required for the implementation of Hodge theory.

A parallel project is to develop abstract commutative and linear algebra as well as group theory and field theory. For large parts, these theories don't need the real numbers. As pointed out above, the notion of localization would be convenient to have for the construction of the rational numbers. In general, it would be good to start by developing the basic commutative algebra such as is assumed in the beginning of Hartshorne. Once we have a reasonable control over the basic notions of ring, algebra, module etc. then much of commutative algebra can be done without any new notational worries.

Linear algebra could lead to a rather extensive development of the theory of matrices, matrix groups, and group representations.

Again in parallel, we need the notions of presheaf and sheaf over a topological space. Here it seems economical to take the abstract point of view and develop directly the notion of sheaf on a Grothendieck site. Along the way the notion of presheaf on a category yields the notion of simplicial set which forms the basis for homotopy theory. The notion which is directly useful for starting out is that of sheaf on a topological space, so we need a construction going between a space and its corresponding Grothendieck site of open sets. It is unclear whether we would prefer using the full abstract notion of point for a site (or maybe a topos?—but with topos theory we may run into universe problems) or whether it is better just to look at the germ for a presheaf on a topological space. In any case, here some *ad hoc* argumentation is necessary, because we would like the germ of a presheaf to be endowed with the same type of structure as the presheaf, in particular operations `plus`, `times` (which work reasonably well because they are just binary operations) but also `mult` for a sheaf of modules (in which case we have to take into account what is the sheaf of scalars), furthermore if the sheaf has a graded or simplicial structure we would like to conserve that, etc; the variety of different types of structures we want to preserve means that it is probably more difficult to come up with a single general theory than to just do each one separately.

Up to here, most of what is listed above has already been treated in one way or another, in at least some user-contributions for at least some of the systems listed in the bibliography (we leave it to the reader using internet to detail the complicated statement of references which would be needed to account for all of that). Left open is the problem of coalescing everything together.

Once we have these various theories, we can put them together to obtain the notion of locally ringed space modelled on a certain type of model space. Depending on which model spaces we look at, we can obtain the notions of differentiable manifold, complex analytic manifold, or algebraic variety (scheme). At this point, perhaps one of the first (and simplest, because it is abstract) things to do is to do the comparison constructions going from schemes (over $\mathbf{C}$) to complex manifolds to differentiable manifolds. We should then be able to construct projective space, projective subvarieties, various differentiable manifolds, and so on. Another thing to do will be exterior algebra and differential calculus over a manifold (again with the comparisons).

It would be interesting to try things out by applying this to the case of Lie groups, and seeing how far into the theory one can go.

Integration is probably the next place where we will run into a major notational problem. This is because what we mean when we say to "integrate" spans a highly diverse and notationally wide-ranging collection of ideas: there is simple Riemann (or Lebesgue) integration on the real line, but also more general integration of a measure, or integration of a differential form over various things such as a manifold, or a smooth chain; or finally integration of currents and the like. The difficulty is that these different notions have a very high degree of overlap, however they are not quite the same, and they require differing notations and even differing levels of notational complexity. I don't currently have any particular ideas to propose to overcome this difficulty.

With the theory of integration would come the possibility of getting at the fundamental properties of analysis on manifolds. At this point we will need the theory of Hilbert and Banach spaces, and we should be able to do the basic theorems such as Stokes' theorem, the Poincaré lemma and de Rham's theorem, Hartogs' theorem for complex manifolds, the mean value theorem for harmonic functions, which gives the maximum principle and unicity of analytic continuation, the representation (and regularity) of cohomology by harmonic forms, leading up to the Hodge decomposition for the cohomology of a complex K"ahler manifold (to take an example from my own research interests). At this stage we will also want to have the basic notions of homotopy theory available.

Once all of the above is well in hand, we would be able to state and prove a first theorem which is a really nontrivial application of this wide swath of mathematics: the fact that the first (or any odd) Betti numbers of complex algebraic varieties are even.

We should then be able to start looking at more advanced things going toward current research problems. This could include for example the construction of moduli spaces of vector bundles, connexions and so forth. Or the investigation of existence problems for connexions over higher dimensional varieties which could at first be viewed just as matrix problems in linear algebra.

Some day in the not-too-distant future we will have a computer-verified proof that the curvature of the classifying space for variations of Hodge structure in the horizontal directions is negative, and all that it entails.

References

[1] P. Aczel. The type theoretic interpretation of constructive set theory. *Logic colloquium 77*, Macintyre, Pacholsky and Paris eds., Springer (1977).

[2] G. Alexandre. An axiomatisation of intuitionistic Zermelo-Fraenkel set theory. See
`http://coq.inria.fr/contribs-eng.html`.

[3] T. Altenkirch, C. McBride. Generic Programming Within Dependently Typed Programming. To appear, WCGP 2002.
`http://www.dur.ac.uk/c.t.mcbride/generic/`

[4] Y. Bertot. Coq'Art. Book in preparation.
`http://www-sop.inria.fr/lemme/Yves.Bertot/coqart.html`

[5] H. Boom. Message on [69], Feb. 16, 2001.

[6] V. Capretta. Universal Algebra in Coq.
http://www-sop.inria.fr/Venanzio.Capretta/universal_algebra.html

[7] L. Chicli, L. Pottier, C. Simpson. Mathematical quotients and quotient types in Coq. in H. Geuvers, F. Wiedijk, eds. *Types for Proofs and Programs*, Springer LNCS **2646** (2003), 95-107.

[8] T. Coquand. An analysis of Girard's paradox. *Proceedings of LICS*, IEEE press (1985).

[9] J. Courant. Explicit universes for the calculus of constructions. In V. Carreño, C. Muñoz, and S. Tahar, eds., *Theorem Proving in Higher Order Logics 2002*, Springer LNCS **2410**, 115-130.

[10] A. Cuihtlauac, Reflexion pour la reecriture dans le calcul de constructions inductives. Thesis, Dec. 18, 2002.

[11] P. Deligne, J. S. Milne, A. Ogus, K. Shih. *Hodge Cycles, Motives, and Shimura Varieties.* Springer LNM **900** (1982).

[12] S. Feferman. Typical ambiguity: trying to have your cake and eat it too. To appear in the proceedings of the conference Russell 2001, Munich, June 2-5, 2001.

[13] M. Fiore, G. Rosolini. Two models of synthetic domain theory. *J. Pure Appl. Alg.* **116** (1997), 151-162.

[14] B. Fitelson, D. Ulrich, L. Wos. XCB, the last of the shortest single axioms for the classical equivalential calculus, cs.LO/0211015, and Vanquishing the XCB question: the methodology discovery of the last shortest single axiom for the equivalential calculus, cs.LO/0211014.

[15] H. Friedman. Finite functions and the necessary use of large cardinals. *Ann. of Math.* **148** n.3 (1998), 803-893.

[16] D. Gabai, G. R. Meyerhoff, N. Thurston. Homotopy hyperbolic 3-manifolds are hyperbolic. *Ann. of Math.* **157** (2003), 335-431.

[17] H. Geuvers. Inconsistency of classical logic in type theory.
http://www.cs.kun.nl/~herman/note.ps.gz
See also other publications at http://www.cs.kun.nl/~herman/pubs.html

[18] H. Geuvers, H. Barendregt. Proof Assistants using Dependent Type Systems, Chapter 18 of the Handbook of Automated Reasoning (Vol 2), eds. A. Robinson and A. Voronkov, Elsevier (2001), 1149 - 1238.

[19] H. Geuvers , F. Wiedijk , J. Zwanenburg , R. Pollack , H. Barendregt. A formalized proof of the Fundamental Theorem of Algebra in the theorem prover Coq, Contributions to Coq V.7, April 2001, `http://coq.inria.fr/contribs/fta.html`

[20] J.-Y. Girard. Interpretation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieure. Thèse d'Etat, Université Paris 7 (1972).

[21] T. Hales. The Flyspeck Project Fact Sheet. `http://www.math.pitt.edu/~thales/flyspeck/index.html`

[22] J. Harrison. Formalized mathematics. (1996). `http://www.cl.cam.ac.uk/users/jrh/papers/form-math3.html`, see also an `html` version at `http://www.rbjones.com/rbjpub/logic/jrh0100.htm`

[23] H. Herbelin. A program from an A-translated impredicative proof of Higman's Lemma. User-contribution in [52], see `http://coq.inria.fr/contribs/higman.html`

[24] R. Holmes. Undefined terms, and the thread of messages following it (in particular J. Harrison's reply), [68] Volume 3, May 1995.

[25] A. Hohti. Recursive synthesis and the foundations of mathematics. `math.HO/0208184`.

[26] L. S. van Bentham Jutting. Checking Landau's "Grundlagen" in the AUTOMATH system. Thesis, Eindhoven University of Technology (1977).

[27] H. Kitoda. Is mathematics consistent? `math.GM/0306007`

[28] L. Lamport. Types considered harmful, or Types are not harmless. This appeared under the first title in a posting by P. Rudnicki on [68], Volume 2, Aug. 1994. A revised version with the second title appeared as a technical report. A balanced discussion presenting both points of view is in the next reference.

[29] L. Lamport. L. Paulson. Should Your Specification Language Be Typed? *ACM Transactions on Programming Languages and Systems* **21**, 3 (May 1999) 502-526.
Also available at
http://research.microsoft.com/users/lamport/pubs/pubs.html

[30] Z. Luo. An extended calculus of constructions. Thesis, University of Edinburgh (1990).

[31] M. Maggesi. Proof of JMeq_eq, see posting on [69], Oct. 17th 2002, http://www.math.unifi.it/ maggesi/coq/jmeq.v

[32] P. Martin-Löf. *Intuitionistic Type Theory*, Studies in Proof Theory, Bibliopolis (1984).

[33] W. McCune, R. Veroff. A short Sheffer axiom for Boolean algebra.
http://www.cs.unm.edu/~moore/tr/00-07/veroffmccune.ps.gz
http://www.cs.unm.edu/~veroff/, http://www.mcs.anl.gov/~mccune

[34] J. McKinna. Reply to thread "How to prove two constructors are different", on [69], Oct. 6th 2003.

[35] I. Moerdijk, S. MacLane. *Sheaves in Geometry and Logic*, Springer (1992).

[36] D. Nowak. Ensembles and the axiom of choice, on [69], Nov. 25th 1998.

[37] R. O'Connor. Proof of Gödel's first incompleteness theorem
http://math.berkeley.edu/~roconnor/godel.html

[38] QED manifesto.
http://www-unix.mcs.anl.gov/qed/manifesto.html

[39] C. Schmidhuber. Strings from logic. CERN-TH/2000-316, hep-th/0011065.

[40] D. Scott. Domains for denotational semantics. *Automata, languages and programming (Aarhus, 1982)*, Springer LNCS **140** (1982), 577-613.

[41] I. Shimada. Vanishing cycles, the generalized Hodge conjecture, and Gröbner bases. math.AG/0311180.

[42] A. Simpson. Computational adequacy in an elementary topos. *Proceedings CSL '98*, Springer LNCS **1584** (1998), 323-342.

[43] C. Simpson. The following files are attached to the source file of the present paper with an `\invisible{ ... }` command. The first three form a unit, the last two compile separately.
(i) `groundwork7.v` (ii) `notation9.v` (iii) `order2.v`
(iv) `qua.v` (v) `h1a.v`
See `http://math.unice.fr/ carlos/inProgress.html`

[44] S. Simpson, ed. *Reverse Mathematics 2001*, to appear.

[45] T. Streicher. Lifting Grothendieck universes (with M. Hofmann); and Universes in toposes. Preprints available at
`http://www.mathematik.tu-darmstadt.de/~streicher/`

[46] M. Wenzel, F. Wiedijk. A comparison of Mizar and Isar. *J. Automated Reasoning* **29** (2002), 389-411.

[47] B. Werner. Sets in Types, Types in Sets. *Theoretical aspects of computer software (Sendai 1997)*, Springer LNCS **1281** (1999), 530-546.
`http://pauillac.inria.fr/ werner/publis/zfc.ps.gz`

[48] B. Werner. An encoding of Zermolo-Fraenkel Set Theory in Coq: see `http://coq.inria.fr/contribs-eng.html`.

[49] B. Werner. Une théorie des constructions inductives. Thèse d'Etat, Univ. Paris 7 (1994).

[50] ACL2 system:
`http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html`.

[51] Alfa system (formerly ALF):
`http://www.math.chalmers.se/~hallgren/Alfa/`

[52] Coq system: `http://coq.inria.fr/`,
especially the reference manual : `http://coq.inria.fr/doc/main.html`.

[53] Helm Coq-on-line library (University of Bologna):
`http://www.cs.unibo.it/helm/library.html`

[54] HOL system:
http://www.afm.sbu.ac.uk/archive/formal-methods/hol.html.

[55] IMPS system, see particularly the theory library:
http://imps.mcmaster.ca/.

[56] Isabelle system :
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[57] The Elf meta-language.
http://www-2.cs.cmu.edu/~fp/elf.html

[58] LEGO system :
http://www.dcs.ed.ac.uk/home/lego/.

[59] Mizar system :
http://mizar.uw.bialystok.pl/.

[60] D. J. Melville. Sumerian metrological numeration systems.
http://it.stlawu.edu/~dmelvill/mesomath/sumerian.html

[61] Nuprl system:
http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html.

[62] The PhoX proof assistant:
http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI/af2.html

[63] PVS (Proof Verification System)
http://pvs.csl.sri.com/

[64] TPS (Theorem Proving System)
http://gtps.math.cmu.edu/tps.html

[65] Z/EVES system, see particularly the Mathematical Toolkit:
http://www.ora.on.ca/z-eves/.

[66] Formal Methods web page at Oxford (this has a very complete listing of items on the web concerning formal methods):
http://www.afm.sbu.ac.uk/.

[67] F. Pfenning. Bibliography on logical frameworks (449 entries!) `http://www-2.cs.cmu.edu/~fp/lfs-bib.html`

[68] The QED project (see particularly the archives of the QED mailing list, volumes 1-3): `http://www-unix.mcs.anl.gov/qed/`

[69] Coq-club mailing list archives: `http://pauillac.inria.fr/coq/contacts-eng.html`.

[70] *Journal of Formalized Mathematics* : `http://mizar.uw.bialystok.pl/JFM/`.

[71] arXiv e-Print archive `http://arXiv.org/`

[72] MathSci Net. `http://www.ams.org/mathscinet` (by subscription)

[73] The Google search engine. `http://www.google.com`