THE `mathz` PROJECT

ARTHUR RYMAN

Abstract. The `mathz` project is my personal effort to create a library of formal specifications for standard mathematical objects, built on top of Z Notation and its mathematical toolkit. I intend to use this library as the basis for my further formalization of topics in mathematics, physics, and software, especially software for computational theoretical physics. I hope that this library will improve both my productivity, through greater reuse of definitions, and my confidence in the correctness of statements, through automated type checking and proof checking. This article discusses the motivation, goals, and structure of the project.

Contents

Date: December 23, 2022.

1. Introduction

When developing complex software, I frequently find myself thinking that it would be a good idea to formally specify it, and to prove that my implementation satisfies the specification.

Of course, formal specification is overkill for the vast majority of software. However, some software does perform complex processing, and so developing a formal specification for it has the potential benefits of both clarifying concepts and seeing how the code can be safely optimized.

Similarly, when reading mathematics, I often think that formalizing it would be a useful activity. The simple exercise of formally stating definitions and theorems would certainly help the learning process. The more advanced task of formally proving theorems would ensure that all implicit assumptions have been made explicit, and that all proofs are in fact correct.

Software and mathematics fruitfully intersect when it comes to computational physics. Wouldn't it be cool to have formal specifications that connect published papers to running implementations? And wouldn't it be even cooler if there was a widely-used, standard specification language so that new results could seamlessly build on prior results?

The `mathz` project is my attempt to achieve this vision. This article describes my plan, technology choices, and results to-date.

2. The `mathz` Library

The core component of the `mathz` project is library of formal definitions for standard mathematical objects and theorems. I have selected LaTeX as the source format and Z Notation as the formal specification language. I'll justify these technology choices below.

2.1. Library Overview.

This section presents an overview of the current structure of the `mathz` library, including a brief description of each article and its dependencies on prior articles.

`core`: This article defines formal generic parameters and arbitrary sets. It does not depend on any other articles.

`categories`: This article defines categories, functors, natural transformations, and related objects. It depends on `core`.

`sets`: This article defines families of sets, bits, constant functions, indicator functions, and supports. It depends on `core`.

groups:

integers:

real_numbers:

complex_numbers:

vector_spaces:

topological_spaces:

manifolds:

lie_groups_and_algebras:

In addition, the library contains the following articles.

project: This article.

formal_specifications:

article_template:

2.2. **Toolchain.** The source code for the `mathz` library is then simply a collection of LaTeX files that Z Notation specifications for mathematical objects. These source files are processed in the following ways:

- Typeset the articles using LaTeX to produce publication-quality PDF documents. Optionally convert the LaTeX into automatically hyperlinked HTML and publish them on the web.
- Type check the articles using *f*UZZ to remove easily-detected type errors and improve the global self-consistency of the article. Note that this is a good start but not the end of the road. A Z Notation document that has no type errors may contain other logical errors. This is where a proof obligation generator and checker is required.
- Extract the formal definitions from the articles so they can be included in other articles that build on them. Technically, this is accomplished by collecting all the formal definitions into a so-called *prelude* file that is supplied to *f*UZZ along with the article being type checked. Conceptually, a prelude file is like a C programming language header file.

2.3. **Open Source.** The `mathz` library should be an open source repository that allows anyone to contribute. Contributions to the library would, of course, have to type check cleanly and pass other quality controls.

The `mathz` library would only be one component of an ecosystem comprised of many authors. Ideally, this ecosystem should be decentralized in terms of content creation, but centralized in terms of content sharing along the lines of package repositories such as PyPI.

3. Z Notation

Z Notation[**spivey-zrm**] is a *formal* software specification language. Saying that a specification language is formal means that it has a precisely defined syntax and semantics so that a computer program can check useful aspects of the specification's correctness. I use the *f*UZZ[**spivey-fm**] type checker for specifications written using Z Notation.

It is fair to say that formal specification languages never achieved the widespread adoption hoped-for by their proponents. Two often-cited obstacles are cost and skill availability.

Writing a formal specification does incur extra upfront cost. Projects, such as the development of safety-critical systems, can justify this extra upfront cost since it can help avoid much more expensive downstream corrective action. Nevertheless,

this upfront cost objection merits further examination. It is well known that the cost of removing a detect in any system increases as development proceeds. It is far cheaper to correct a defect in a formal specification before any code has been written than in a deployed software system running in production where it could impact thousands of users. I therefore regard the cost objection as dubious.

However, the skill objection is very real. The effective use of a language like Z Notation requires that the author have a certain capacity for mathematical abstraction, a capacity that is not required for the effective use of modern high-level programming languages. Nevertheless, I have personally found Z Notation to be a very useful tool for organizing one's thoughts and have returned to it repeatedly over the years. It therefore seems to be a good idea to keep a specification language as simple as possible and to make it as similar as possible to the standard mathematics that most software developers normally acquire in the course of their education.

The thing I find most appealing about Z Notation is that it has a very firm and simple mathematical foundation. Indeed, one can view Z Notation as a formal language for writing mathematics. However, Z Notation differs from informal mathematical notation in that it is based on *typed set theory*. Recall that typed set theory was invented by Bertrand Russel as a way to avoid certain logical paradoxes that arose in the foundations of mathematics. For example, see Russel's paradox. The use of simple type theory makes Z Notation very appropriate for specifying software where data types are the norm. It also makes Z Notation more checkable. Indeed, specifications written in Z Notation can be type checked, a process that can catch many common errors and omissions.

There is an interesting parallel between mathematics and programming languages with respect to the role of types. Working mathematicians implicitly use set theory which neither defines nor uses the concept of a type. Similarly, the two most popular programming languages today, JavaScript and Python, do not require variables to be declared as being restricted to any specific type. These languages do have data types, such as integers and strings, but a variable can reference an object of any type. Such languages as said to be *dynamically-typed* because the type associated with a variable can change during execution. However, many JavaScript and Python developers appreciate the benefits of type declarations which are required in *statically-typed* languages such as Java, C++, and C#, and this has led to the introduction of type support in both JavaScript and Python. In a statically-typed language, each variable is declared to be of a specific type and is only allowed to reference objects of that type. The TypeScript language is a version of JavaScript that supports type declarations. The Python language has steadily improved its *type hint* support through a sequence of Python Enhancement Proposals, e.g. PEP 484 - Type Hints. The Z Notation language can be thought of as a formal language that adds type declarations to informal mathematics. I think of Z Notation as being a high-level, statically-typed, nonexecutable functional programming language.

The Z Notation system consists of a minimal core language and a small, but useful, *mathematical toolkit*. The core language covers sets, logic, the integers, and type constructors for power sets, Cartesian products, and schemas. The mathematical

toolkit builds on the core language and defines more advanced mathematical objects such as sequences, bags, and functions.

While the mathematical toolkit is adequate for specifying typical software systems, it is missing definitions for more advanced mathematical objects. For example, the set of integers, $\mathbb{Z}$, is built into Z Notation, but rational, real and complex numbers are not. The consequence of this omission is that anyone trying to use Z Notation for specifying more advanced mathematics has to first build up a lot of basic definitions. The main goal of the `mathz` project is to do this work once and then reuse it as the basis for more advanced specifications.

One cosmetic limitation of Z Notation is that it cannot fully reproduce the elaborate notation that mathematicians freely invent. In Z Notation, mathematical operators are either infix, prefix, or postfix whereas in informal mathematics operators may also be superfix, subfix, or surroundfix, not to mention two-dimensional notations such as binomial coefficients, 3j-symbols, 6j-symbols, commutative diagrams, etc. Every domain of mathematics invents its own specialized notations. Mathematicians further abbreviate the notations to speed up the task of writing on paper or blackboard. One can simulate some of these notations through clever use of LaTeX, but perhaps that is not the right goal. Perhaps Z Notation should be kept simple so that the definitions are easy to understand as opposed to being easy to write by hand. One can always augment a Z Notation specification with informal mathematical notation. Another possibility is to use symbolic computation libraries such as SymPy that can render mathematical objects using arbitrary LaTeX.

A more serious shortcoming of Z Notation is that it lacks a proof checker. In fact, Z Notation lacks a means to formally encode a proof. This means that in practice a formal specification written in Z Notation consists primarily of formal definitions and constraints, with perhaps some informal proofs. The lack of a proof checker means that arguably some of the most important content of mathematical writing is outside the scope of Z Notation. However, this omission can be partially mitigated by combining Z Notation with one of the many available proof checkers, some of which, such as Z/EVES, have been successfully used with it. See The Z/EVES 2.0 User's Guide. More on this later.

In summary, the attractiveness of using Z Notation for more advanced mathematics is that it is fairly close to standard mathematical notation but can also be type checked and, by integrating other tools, proof checked. Furthermore, being a formal language allows Z Notation to potentially take advantage of computer tools for indexing and searching content.

4. High-Level Requirements

Here are the main high-level requirements of the `mathz` project:

- Integrated Math Library

  Create an integrated library of formal Z Notation definitions for standard mathematical objects.

- Preserve Seamless Interoperability with LaTeX

Enable the creation of standard LaTeX mathematical articles that use the library.

- Powerful Search and Navigation

  Enable users to easily search and navigate the formal definitions so they can find what they need and can build on them.

- Check Formal Proofs

  Create bridges with proof assistants so that definitions from the standard library can be used in formal proofs.

- Generate and Check Proof Obligations Automatically

  Enhance the automatic type and proof checking of Z Notation specifications by generating all required proof obligations and handing them off to proof assistants. I hope that the current generation of proof assistants is powerful enough to automatically generate the required proofs.

These high-level requirements are elaborated below.

4.1. **Seamless Interoperability with LaTeX.** A major pragmatic difference between proof assistants and Z Notation is that proof assistants define their own source file formats whereas Z Notation is seamlessly integrated with LaTeX. Z Notation can be embedded in LaTeX documents by means of a LaTeX package which means it can be combined with all the other LaTeX packages that mathematicians use every day.

Both Coq and Lean can generate other document formats from their source formats, possibly with some restrictions. Given the dominant role of LaTeX in scientific publishing it is advisable to leave it as the top-level format, rather than generate it from the proof assistant source files.

Given present proof assistant technology, it is unlikely that a long formal proof would usefully be published in a scientific journal, primarily because such proofs are not designed for human-readability as traditional documents. Rather, they are designed for machine processing. A human reader interested in the details of the proof would likely want to access its source files and view them using an interactive tool. It therefore makes sense to keep the formal specification separate from the formal proof. However, the bridge between the specification and the proof should be automatic and verifiable. For example, trusted tools could generate digital signatures for both the specification and its translation into the language of the proof assistant. This would guarantee that the proof assistant was in fact proving the theorems stated in the specification. Perhaps there is a role for block chain here.

Another potential advantage of separating the specification from the proof assistant is that proof assistants are rapidly evolving and it might be useful to reverify a proof in several competing proof assistants, or newer versions of a given proof assistant. Thus using LaTeX with Z Notation might provide a more stable and future-proof format for formalizing mathematics.

4.2. **Powerful Search and Navigation.** A user should be able to quickly determine if a mathematical object has been defined in the library and, if defined, easily navigate to it. All articles in the library should therefore be indexed by a full-text search engine such as Apache Lucene.

While unstructured text search is very useful, we can do better by exploiting the formal syntax of Z Notation. All objects defined using Z Notation can only refer to previously defined objects. This means that all defined objects can be regarded as nodes in an acyclic dependency graph. Therefore, given an object, the user should be able to easily trace back through all of its dependencies. Furthermore, the user should be able to easily trace forward and find all the objects that depend on it.

These requirements can be satisfied through hyperlinks and indexes. Although HTML is the preeminent document format for hyperlinking, PDF also supports it. Suitable hyperlinks can be added to the PDF version of an article using the LaTeX `hyperref` package. Furthermore, there are many tools that can generate HTML from LaTeX.

However, it is error-prone to rely on authors to manually add suitable links and targets to their articles. Therefore, each article should be processed to automatically extract all object definitions and references. Fortunately *f*UZZ can perform such processing as a side-effect of type checking. The *f*UZZ program can generate an easily-parsed data file containing all definitions and references, traced back to their line numbers in the LaTeX source file. This data file can then be used to generate a variety of powerful navigation aids such as dependency graphs, indices, and cross-references.

4.3. **Check Formal Proofs.** The readability of an article is much improved by the inclusion of examples, remarks, theorems, and proofs. Z Notation contains some commands that aid in the formatting of proofs, but these are ignored by the *f*UZZ type checker. It would be useful to develop support that could type check proofs and potentially convert them into Coq or Lean programs so they could be checked.

Note that the statement of examples, remarks, theorems, and other theorem-like content will be type checked by *f*UZZ if they are placed in a `zed` environment. Doing so adds them as axioms to the specification, but if they are true then the meaning of the specification is unchanged since they are a consequence of the other axioms.

To accomplish proof checking, a bridge tool would convert all Z Notation content into the proof assistant language. Propositions enclosed in theorem-like environments would be converted into theorems requiring proofs. Informal proofs, marked up with sufficient detail, would be converted into formal proofs.

4.4. **Generate and Check Proof Obligations Automatically.** As mentioned above, a Z Notation specification that type checks cleanly may still contain errors. As a simple example, consider division by zero. Such an expression type checks cleanly but is meaningless since zero in not is the domain of the division operator.

In general, whenever a Z Notation specification contains a function application there is a corresponding proof obligation that the arguments of the function application are in the domain of the function. In practice, it should be easy to prove that the function application makes sense. However, doing so consistently would be very

tedious for a human. This seems like an ideal situation for a proof checker. All such proof obligations could be automatically extracted. Furthermore, if the proof checker failed to find a proof then this would be a red flag indicating that perhaps the specification is missing some constraints.

There are several other types of proof obligation for Z Notation specifications.

5. Related Approaches

Creating an integrated math library is not a new idea. There are two types of projects underway now. These are *math wikis* and *proof assistants*.

5.1. **Math Wikis.** A math wiki is a collection of web pages that contain informal math articles. The most well-known are:

- Wikipedia
- Wikimath
- Wolfram MathWorld

These websites contain extensive content and are a great resource for anyone looking for definitions and descriptions of mathematical objects. However, because they contain informal mathematics, they don't enable type checking or proof checking of content.

Wikipedia and Wikimath are community-authored and as such suffer from quality control and consistency. It is very common to see the mathematical notation change from paragraph to paragraph on a single page. Furthermore, there is much repetition of content from page to page.

Wolfram MathWorld is a commercial product and is tightly integrated with Mathematica. In a sense it does have a formal language behind it, namely the Wolfram Language, although the language is designed to support symbolic computation as opposed to specification.

5.2. **Proof Assistants.** A proof assistant is a software tool that can automatically check the correctness of a proof written in a formal language, and can help the user, via so-called *tactics* and other interactive features, find a proof of a theorem. Of course, there is no general algorithm for finding the proof of an arbitrary theorem.

There are many proof assistants under development in academia and industry. For a comprehensive comparison see the Proof assistant article in Wikipedia. Proof assistants can be classified according to their choice of mathematical foundation. Broadly speaking, there are three main flavours of foundations:

- Set Theory
- Simple Type Theory, aka Higher-Order Logic (HOL)
- Dependent Type Theory

Dependent type theory reduces theorem proving to type checking, but is less natural as a foundation for working mathematicians. The two most active dependent type theory proof assistants are:

- The Coq Proof Assistant

- Lean Theorem Prover

Both Coq and Lean have extensive libraries of definitions and theorems. Why then should I bother with Z Notation? That's a great question.

I have worked through a lot of the Coq and Lean materials. I am impressed with their power. However, the price you pay is that with dependent type theory you give up the simple notion of a set. On the other hand, you don't have any unstated proof obligations as you do with simple type theory. If a Z Notation specification type checks, it may still have error, e.g. division by zero, or in general applying a function to an element not in its domain. This situation cannot occur in dependent type theory since all functions are total!

Of the two, Lean is more modern, open, and active. Perhaps if I learn more I will see a natural way to bridge between Z Notation and Lean. I will persist with Z Notation a while longer, but am open to the possibility that Lean is a better way to go. In any case, I will investigate using Lean as a proof assistant for Z Notation specifications. As an acid test, I will attempt to formalize homological algebra in Z Notation, and compare it to the Lean library. It appears that Lean has a very effective mechanism for handling generic mathematical notation, namely via Haskell-inspired *type classes*.

That being said, I find Lean proofs to be far less readable than traditional informal proofs. To understand a Lean proof, you really need to explore it in an interactive tool such as the VSCode Lean plugin. The Lean community acknowledges this problem and recognizes a need for a high-level *proof blueprint* that precedes and explains the development of formal Lean proofs. For example, see Blueprint for the Liquid Tensor Experiment.

Patrick Massot developed leanblueprint, a tool for generating blueprints. The leanblueprint tool is itself a plugin for his plasTeX tool, a Python package that processes LaTeXdocuments into an XML-DOM-like object which can be used to generate various types of output. Clearly, one of these types of output is HTML. Refer to the plasTeX Documentation for more detail. Massot created another tool that converts Lean source code into interactive HTML, preserving LaTeX-like mathematical typesetting. Refer to the Lean in LaTeX blog post by Kevin Buzzard. Refer to the GitHub project format_lean for the formatter code.

Perhaps Z Notation would be a superior alternative to informal LaTeX. The documentation for plasTeX gives an example of converting a tikz-cd commutative diagram to HTML so perhaps it could do a decent job on Z Notation. If Z Notation is not supported out-of-the-box, perhaps it would be feasible to create a plasTeX plugin. In any case, I first need to confirm that Z Notation is up to the task of formalizing something nontrivial like homological algebra.

Based on my current understanding of Lean, I feel that it is feasible to embed Z Notation in dependent type theory and, by extension, Lean. After all, simple type theory should be a simple case of dependent type theory. The main insight here is that in Z Notation there is really only one *sort* of term, namely objects, whereas in Lean there are types, elements of types, type universes, and propositions. In Z Notation, an object is either a set or an element of a set. Every object has a type,

but types are themselves regarded as sets, namely maximal sets. Z Notation has no syntax for naming either types or propositions as distinct sorts of objects.

Z Notation can define generic types that depend on other types. For example, Z Notation can define the generic type of sequences of a given type. Such generic types are a form of dependent type. However, Z Notation cannot define types that depend on elements of other types. For example, if n is a given natural number, Z Notation cannot define the type consisting of sequences of length n. In Z Notation, sequences of length n are a subset of the set all sequences, both finite and infinite.

There are a couple of indirect ways to give propositions names in Z Notation. One can use the natural correspondence between sets and propositions, namely the set associated with a proposition is the subset of all elements that satisfy the proposition. One can also define a schema whose declaration part defines the domain of the proposition and whose predicate part defines the conditions for the proposition to be satisfied. However, Z Notation does not support the notion that a proposition is a type and that an element of that type is a proof of the proposition.

There is no formal way to represent a proof in Z Notation, although *f*UZZ does provide a LaTeX environment for informal proofs. One can embed the Z Notation statement of theorems in LaTeX documents by wrapping an axiom in a proof-like environment. Here we use the observation that a theorem is a redundant axiom in the sense that it can be deduced from other axioms. Therefore, adding the statement of a theorem as an axiom contributes no new information to the specification as far as the semantics of the specification is concerned.

One further remark on bridging Z Notation with a proof assistant concerns proof obligations. Any tool that translates Z Notation into the language of a proof assistant must also generate proof obligations that are not checked by *f*UZZ. The main case is that of checking the domain of function applications, but there are several others. For example, consider an expression involving integer division, p/q. *f*UZZ will happily accept this expression as long as it can check that p and q are integers. However, this expression is meaningless when $q = 0$. We therefore have the proof obligation that $q \neq 0$. A tool that could generate proof obligations of Z Notation would be useful in its own right, independly of any bridge to a proof assistant.

6. Conclusion

This document has described the motivation and goals for the `mathz` project. The basic management, search, and navigation requirements can be satisfied through a modest development effort. The addition of proof checking and proof obligation generation require more thinking and investigation and could lead to an abandonment of Z Notation in favour of Lean or some other proof assistant.

Email address, Arthur Ryman: `arthur.ryman@gmail.com`