THE `mathz` PROJECT

ARTHUR RYMAN

Abstract. The `mathz` project proposes to develop a collection of formal specifications for standard mathematical objects that builds on and extends the mathematical toolkit that comes with Z Notation. This library is intended to be used as the basis for the further development of mathematics, physics, and software, especially software for theoretical physics. This article describes the motivation, goals, and structure of the project. The hoped-for benefits of this project are improved productivity through greater reuse of mathematical definitions and greater confidence in the correctness of mathematical and physical theories and their software implementations.

Contents

1. Introduction

Z Notation is a formal specification language invented for the purpose of software development. By *formal* I mean that the language has a precisely defined syntax and semantics so that the *correctness* of a specification written in such a language can be checked, to some degree, by a computer program. I use the *f*uzz type checker for specifications written using Z Notation.

Date: August 14, 2022.

It is fair to say that formal specification languages never achieved widespread adoption by software developers. The use of a formal specification language imposes additional cost at the start of a project and the pool of developers who have the required skills is limited. Because of these challenges, the use of formal specification languages has been confined to projects where the additional upfront cost can be justified, for example in the development safety-critical systems where downstream errors are very expensive to resolve. 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.

The upfront cost objection merits further examination. It is well known that the cost of removing a detect 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. 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 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. This point makes Z Notation more suitable 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 the Zermelo-Fraenkel axioms for set theory which neither defines nor uses the concept of a type. Similarly, the two most popular programming languages today are JavaScript and Python both of which do have data types but neither of which require variables to be declared as being of any specific type. However, many JavaScript and Python developers appreciate the benefits of type declarations which are required in languages such as Java, C++, and C#, and this has led to the introduction of type support in JavaScript and Python. The TypeScript language is a version of JavaScript that supports type declarations. Python has steadily improved type hint support through a sequence of Python Enhancement Proposals, e.g. PEP 484 - Type Hints. It is fair to say that Z Notation is a formal syntax for informal mathematics together with type declarations. I think of Z Notation as being a statically-typed, nonexecutable functional programming language.

The Z Notation system consists of a core language and a small, but useful, *mathematical toolkit*. The core language covers the integers, basic logic, sets, 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 build up a lot of basic definitions first. 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 their articles. In Z Notation mathematical operators are either infix, prefix, or postfix whereas in informal mathematics operators may also be *super*-fix, *sub*-fix, or *surround*-fix not to mention two-dimensional notations such as binomial coefficients, 3j symbols and 6j symbols. One can simulate some of these notations through clever use of LaTeX but perhaps that is not the point. Perhaps Z Notation should be kept simple so that the definitions are easy to understand as opposed to being easy to write. 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 formal proof checker. 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 built-in proof checker means that arguably some of the most important content of mathematical writing is outside the scope of Z Notation. However, this omission could 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, potentially, proof checked. Furthermore, being a formal language allows Z Notation to potentially take advantage of computer tools for indexing and searching content.

2. Goals

Here are the main goals of the `mathz` project:

- Integrated Math Library

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

- 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 checkers so that definitions from the standard library can be used in formal proofs.

- Generate and Check Proof Obligations Automatically

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

These goals are elaborated below.

3. Integrated Math Library

This is not a new idea. There are two types of similar efforts underway now. These are *math wikis* and *proof assistants*.

3.1. **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, they don't enable type checking of mathematical documents.

Wikipedia and Wikimath are community-authored and as such suffer from quality control and consistency. It is very common to see the notation change from paragraph to paragraph on a single page. Furthermore, there is much repetition and redundancy 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, although the language is designed to support symbolic computation as opposed to specification.

3.2. **Proof Assistants.** There are many proof assistants under development in academia. The two most relevant ones are:

- The Coq Proof Assistant
- Lean Theorem Prover

The main theoretical difference between these two and Z Notation is that they support *dependent type theory* in contrast to *simple type theory*. Dependent type theory enables more effective theorem proving but is less natural for working mathematicians. 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 Lean materials. I am impressed with its power. However, the price you pay is that you give up the simple notion of a set. 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.

3.3. **The mathz Library.** The mathz library is then simply a collection of LaTeX source files. These source files are processed in the following ways:

- Typeset the articles using LaTeX to produce publication-quality PDF documents.
- 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 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.

The mathz 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.

4. Seamless Interoperability with LaTeX

Another pragmatic difference between these proof assistants and Z Notation is that they define their own file formats whereas Z Notation plays nicely with LaTeX. Z Notation is implemented as a LaTeX style 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 formats, possibly with restrictions. Given the dominant role of LaTeX in scientific publishing it would safer to leave it as the top-level format.

5. 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.

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. This data file can then be used to generate a variety of powerful navigation aids such as dependency graphs, indices, and cross-references.

6. 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*UZZtype 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.

7. Generate and Check Proof Obligations Automatically

As mentioned above, a Z Notation specification that type checks cleanly may still contain errors. As a simply 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.

8. 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`