THE ROLE OF FORMALIZATION AND Z Notation

ARTHUR RYMAN

Abstract. The notion of *type* was introduced into set theory by Russel in an effort to establish a firm foundation for mathematics. Types are now a standard feature of many computer programming languages. Z Notation is a formal language, based on *typed set theory*, for specifying computer programs. Since Z Notation is a formal language, a Z specification can itself be programmatically checked for *type errors*.

Writing valid Z specifications requires that the author explicitly define all terms. This encourages a style in which complex terms are gradually built up from simpler, previously defined terms. This article proposes to use Z Notation to formally specify mathematical structures, even if there is no intention of using them in computer programs. The hoped-for benefits of doing so are that the exercise of formalization will lead the author to a better understanding of the subject matter and result in a clearer, higher quality, specification.

Contents

1. Types

Computer languages that support the notion of data types can be divided into two major groups, namely those that are *statically typed* and those that are *dynamically typed*.

Date: February 11, 2025.

Statically-typed languages, such as Java, C++, and C#, have rich type systems and require that all variable declarations and expressions have well-defined types which can be determined by statically analyzing the source code. In a statically-typed language, the type of a variable cannot change during program execution.

Dynamically-typed languages, such as Python and Javascript, also have rich type systems but allow greater flexibility in the use of variables and expressions. In these languages, a variable may be assigned an expression of any type. Furthermore, the same variable may be reassigned an expression of a different type elsewhere in the program. In general, the type of an expression depends on the types of its input variables. The type of a variable may therefore change during the course of program execution.

Although no program can tell you if an arbitrary program is correct, a compiler for a statically-typed language can at least detect type errors. Correcting errors at compile-time is often much easier than doing so at execution-time. Z Notation can be thought of as a high-level, nonexecutable, statically-typed, functional programming language.

There are many similarities between mathematics and programming. Both require the definition and use of named objects that are built up from simpler objects. The concept of type applies to both. Bertrand Russell discovered typed set theory in his efforts to resolve some paradoxes in the foundations of mathematics long before computers were a reality. Theoretical computer science now makes extensive use of types.

Specifications act as a bridge between requirements and programs. Mathematics can be used to precisely specify the requirements for programs. Z Notation is a formal specification language based on solid mathematical foundations. However, Z Notation can be used to formalize mathematics even if the goal is not to program a computer.

2. Formalization

Z Notation has the following desirable characteristics that make it useful for formalizing mathematics:

- It is based on solid mathematical foundations, namely typed set theory.
- It is expressive enough to describe common mathematical structures.
- It is a formal language and has an efficient type-checker, namely *f*uzz.
- It has a syntax that allows it to be embedded in LaTeX documents so authors can use their normal authoring tools.

I have used Z Notation to write formal specifications for several computer applications. One of the largest was the W3C WSDL 2.0 specification. Using Z Notation was an effective way to expose missing definitions and inconsistent usages in the specification.

I have also used Z Notation to formalize mathematics and have found that this exercise is a great way to make sure that all concepts are well-defined. This is especially

helpful, although very labour-intensive, when dealing with complex mathematical structures.

I believe that the effort of precisely defining every needed concept pays off. Consider how Cédric Villani [4] described the mathematical writing style of Alexander Grothendieck:

> Let me quote a famous mathematical text by Alexander Grothendieck, one of the most famous mathematicians of all times, who wrote long writings. In one of them he talks about the metaphor *de la noix*. You know the parable, the metaphor, of the nut to explain the difference between his style and the style of his fellow mathematician Jean-Pierre Serre. Both of them working in the same area of mathematics, but very different styles. Grothendieck said, "Imagine that we have a nut to open. The style of Serre would be take a hammer and, Bang!, smash the nut. My style would be to take the nut and put it in a sea of acid so that it would be dissolved very slowly, the crust of the nut, without noticing anything." And yes, experts tell us that the style of Grothendieck is like everything is very incremental from one step to another to another by very tiny steps so that you have the impression that nothing occurs and we are really making no progress, and at the end it's proven, there it is, the big theorem is done.

Similarly, Pierre Deligne [1] said:

> From Grothendieck I have also learned not to take glory in the difficulty of a proof: difficulty means we have not understood. The ideal is to be able to paint a landscape in which the proof is obvious. I admire how often he succeeded in reaching this ideal.

The message is clear: a slow, methodical build-up of concepts is a good thing in mathematics.

I am therefore going to systematically evaluate Z Notation as a mathematical specification language, primarily as a way to take notes. Mathematical definitions build on one another so it's important to carefully structure a set of specifications as a unified library that avoids unnecessary duplication. I am calling this library `mathz`.

3. The Zen of Z

Based on my efforts to date, I have had several epiphanies about how to use Z Notation for mathematics. Here is a list, which I expect to grow over time as I get more experience.

- Use schemas for *situations*. These arise as the contexts for definitions or theorems. For example, "Let x and y be positive integers such that $x < y$."
- Use tuples for *structures*. For example, "A *group* is a pair $(G, *)$ consisting of a set of elements G and a binary operation $*$ on them that satisfies certain axioms."
- Use systematic schema decoration to get copies of situations rather than schema renaming. Decoration is effortless. Renaming requires work.

- Use prime decorations when composing situations.
- Reuse globally-defined symbols in schemas when they have the required class, arity, and precedence. Except when you actually need to refer to the globally-defined symbol within the schema.
- Never define a new symbol that uses superscripts since that conflicts with decoration and will cause a LaTeX error.

TODO: expand on these rules

4. Checklist for Formalizing Structures

Certain standard items should be defined when formalizing a structure such as a group, ring, or topological space. A central theme of modern mathematics is that structures are the objects of some category. This means that in addition to defining the structure, we also need to define the morphisms between them.

In fact, it is more proper in some sense to regard a category as being a category of morphisms rather than a category of objects since the objects can be recovered from the morphisms. Also, a given set of objects may have more than one interesting sets of morphisms. For example, consider the set of topological spaces. In this case we can define the morphisms to be either the set of continuous maps or the set of homotopy classes of continuous maps.

Here is a recommended checklist of items to define with a suggested order and naming scheme. This is illustrated using the example of groups.

(1) Define a schema for an object, e.g. $Group[\mathsf{t}]$. Note that we give somewhat arbitrary names to schema components in order to simplify reference to them and make constraints on them easy to express and understand. In general, the defined object may depend on one or more generic parameters. For example, a structure may have one or more sets of elements so we need to provide generic parameters from which to draw these sets. These generic parameters can be thought of as defining a universe of discourse for the objects.

(2) Define the set of objects corresponding to the generic parameters, e.g. $group[\mathsf{t}]$ is the set of all groups in t. The elements of this set are usually tuples, e.g. $(G, (\_ \cdot \_))$. By mapping the schema to a tuple we have erased the component names and instead now must refer to components by their position in the tuple.

(3) Define a function that maps sets to objects on that set. For example, $groups\_on(G)$ is the set of all groups on G.

(4) Define a schema for a morphism, e.g. $Group\_Hom$. Morphisms must include their source and target objects.

(5) Define the set of all morphisms, e.g. $group\_Hom$.

(6) Define the function that maps a pair of objects to the set of morphisms between them, e.g. $group\_hom$

(7) Define a schema for the composition of two morphisms, e.g. $Group\_Composition$.

(8) Remark that the composition of morphisms is a morphism.

(9) Remark that composition is associative.

(10) Define a schema for an identity morphism, e.g. *Group_Id*.

(11) Define the function that maps an object to its identity morphism, e.g. *group_id*.

(12) Remark that the identity morphism is a left and right identity element with respect to composition.

(13) Define a schema for what it means to be a subobject, e.g. *Group_Subgroup*. This may not always make sense.

(14) Define the set of subobjects of an object, e.g. *subgroup*.

(15) Define the map from a subobject to its inclusion morphism, e.g. *Group_Inclusion*.

(16) Define a schema for the image of a morphism, e.g. *Group_Image*.

(17) Remark that the image of a morphism is a subobject.

(18) Define a map from morphisms to their images, e.g. *group_image*.

(19) Define a schema for the kernel of a morphism, e.g. *Group_Kernel*.

(20) Remark that the image of a morphism is a subobject.

(21) Define a schema for what it means for a subobject to induce a quotient, e.g. *Group_Normal*. For groups, the subgroup must be normal. For rings, the subring must be an ideal.

(22) Remark that the kernel of a morphism is normal for groups, an ideal for rings, as appropriate.

(23) Define the set of all normal subobjects, e.g. *group_normal*.

(24) Define a schema for the quotient of an object by a normal subobject, e.g. *Group_Quotient*.

(25) Define a function that maps a normal subobject to the quotient, e.g. *group_quotient*.

(26) Define a schema for the projection morphism of a normal subobject onto its quotient, e.g. *Group_Projection*.

(27) Define a function that maps a normal subobject to its projection, e.g. *group_projection*.

(28) Carry on for cokernels and coimages as appropriate.

5. Definitions and Notations

Use descriptive names when defining objects and use consistent wording. For example, "Define *group*[t] to be the set of all groups in t."

Reserve the term *denote* for defining notation and always define the notation as syntactic sugar for a previously defined and well-named object. For example, "Let $x * y$ denote the group operation $op(x, y)$."

6. Given Sets and Signatures

Although the integers are built into Z Notation, the real numbers are not. In principle, one could first build up the rational numbers from the integers, and then the real numbers from the rational numbers, but that would take a lot of time and not really accomplish much. I'll assume that the real numbers are sufficiently well-understood and do not need a complete formalization. Instead, I'll define real numbers as a given set and then declare the types and signatures of the usual constants and operations of real arithmetic.

7. Names and Symbols

Integers and reals are distinct types in the sense of typed set theory. But distinct types are disjoint sets. This implies that the 0 element of the integers is not the same object as the 0 element of the reals. In fact, they aren't even comparable within a Z specification since doing so is a type error and the specification would therefore not be valid. The name `0` is built-in to Z Notation as the name of the 0 element of the integers. A Z specification must therefore introduce a new name to refer to the 0 element of the reals. However, working mathematicians regard the integers as a subset of the reals and therefore have no need to deal with duplicate names. As Henri Poincaré [2] said:

> I think I have already said somewhere that mathematics is the art of giving the same name to different things. It is enough that these things, though differing in matter, should be similar in form, to permit of their being, so to speak, run in the same mould.

How then can we make a Z specification valid while still honouring normal mathematical practice? The way out of this difficulty is that if a mathematician would normally regard two distinct objects as the same then we will display them using the same symbol but give them distinct Z names. To paraphrase Poincaré:

> Formalizing mathematics is the art of using the same symbol, but giving different names to different things. It is enough that these things, though differing in type, should be similar in typography, to permit of their being, so to speak, run in the same mould.

For example, Mike Spivey [3] describes three symbols defined in the *f*uzz package that are each used to display two distinct objects:

> A few symbols have two names, reflecting two different uses for the symbol in Z:
>
> - The symbol `⨾` is called `\semi` when it is used as an operation on schemas, and `\comp` when it is used for composition of relations.
> - The symbol \ is called `\hide` as the hiding operator of the schema calculus, and `\setminus` as the set difference operator.
> - The symbol ↾ is called `\project` as the schema projection operator, and `\filter` as the filtering operator on sequences.

> Although the printed appearance of each of these pairs of symbols is the same, the type checker recognizes each member of the pair only in the appropriate context.

This approach keeps both the Z type checker and the mathematician reader happy. For example, in my formalization of the reals:

- The symbol 0 is named `0` when it is used as an integer, and `\zeroR` as a real number.
- The symbol + is named `+` as the addition operator on the integers, and `\addR` as the addition operator on the reals.

The mathematician reader will always be able to infer the actual type of an object from its context.

8. Theorems, Examples, and Remarks

In mathematical writing it is good practice to follow a definition with some examples. When an example is given, there is a proof obligation associated with it, namely that whatever the author is asserting about the example is in fact true. Similarly, when the author makes a remark or states a theorem, there is also a proof obligation. In each of these cases, the author makes a statement and asserts it to be true, but may not offer any proof.

Z Notation allows the author to add constraints to a specification. However, if the constraint is a logical consequence of the preceding definitions and constraints then it adds nothing to the specification. Therefore, we can freely add examples, remarks, and theorems to a specification without changing its meaning. These statements should be placed in a `\zed` box that is enclosed in an appropriate LaTeX theorem-like environment.

For example, the following statement is true and therefore does not change the meaning of the specification but will be type-checked by *f*uzz, .

Example.

$1 + 1 = 2$

The benefit of placing theorem-like statements in `\zed` boxes is that the *f*uzz type checker will check their type-correctness.

8.1. **Generic Theorem-Like Statements.** It will often be the case that a theorem-like statement will involve generic constructs. Although Z Notation provides a mechanism for introducing formal generic parameters into the definition of generic constants and schemas, it does not provide such a mechanism for constraints. We can work around this limitation by declaring a set of global given sets which are understood to be used in theorem-like statements where they are taken to represent arbitrary sets. The article on topological spaces introduces the global given sets X, Y, and Z for this purpose.

For example, we can remark that every element x of an arbitrary set X is a member of that set.

Remark.

$\forall\, x : \mathsf{X} \bullet x \in \mathsf{X}$

8.2. **Hyperlinks.** Specifications refer to other specifications. There two ways to create hyperlinks, namely `\href` and `\url`. The behaviour of these links differs by viewing application. I use three PDF viewers, namely Preview, TeXShop, and Acrobat Reader. IntelliJ also has a PDF viewer but I rarely use it.

The `\href` links generally work within the same document, although the Preview app has file permission issues when linking between documents.

The free Adobe Acrobat Reader app works better than Preview. It can open `\href`-linked local documents. It launches a brower for `\url`-linked documents.

Testing hyperlinks to another PDF document:

```
\url{https://agryman.github.io/mathz/complex-numbers.pdf}
```

https://agryman.github.io/mathz/complex-numbers.pdf

```
\url{mathz-formal-specifications.pdf}
```

mathz-formal-specifications.pdf

```
\url{complex-numbers.pdf}
```

complex-numbers.pdf

```
\url{./complex-numbers.pdf}
```

./complex-numbers.pdf

9. Simulating Standard Mathematical Notation

The expression (a, b) usually denotes an ordered pair in mathematical writing. Sometimes an author asks us to interpret (a, b) as an open interval of the real number line. In this case we recognize that the symbols used in the expression (a, b) do not have their usual meanings and we interpret them accordingly. The notation (a, b) for denoting an open interval of the real number line is arguably very compact, convenient, and easy to understand. In contrast, the Z Notation function application interval(a, b), defined in the article on real numbers, may strike the reader as somewhat verbose and clumsy, albeit precise.

9.1. **Prefix, Infix, and Postfix Operators in fuzz.** There is a way to achieve some of the compactness of standard mathematical notation in Z while preserving its precision and type-correctness. The approach is to take advantage of the *f*uzz type checker's ability to define infix and postfix operator symbols. In the grammar used by *f*uzz, postfix operators have higher precedence than prefix operators, and prefix operators have higher precedence than infix operators. Infix operators are assigned numerical priorities of 1 through 6 with higher priorities taking precedence over lower ones. We can use these relative precedences to define operators that allow us to construct expressions that resemble usual mathematical writing.

9.2. **Defining Intervals with Operators.** First, we use LaTeX bold styling to provide a visual cue that helps the reader distinguish the interval $(\!|\, a \,,\, b \,|\!)$ from the usual ordered pair (a, b). Next, we interpret the symbols as prefix, infix, and postfix operators. The symbol $(\!|$ is a prefix operator that maps a to $(\!|\, a$, the open interval bounded below by a. Similarly, $|\!)$ is a postfix operator that maps b to $b \,|\!)$, the open interval bounded above by b. Finally, we interpret the symbol $\,,\,$ as an infix operator that forms the intersection of the two intervals.

$$
\begin{array}{l}
(\!| : \mathbb{R} \longrightarrow \mathbb{P}\,\mathbb{R} \\
\_ \,|\!) : \mathbb{R} \longrightarrow \mathbb{P}\,\mathbb{R} \\
\_ \,,\, \_ : \mathbb{P}\,\mathbb{R} \times \mathbb{P}\,\mathbb{R} \longrightarrow \mathbb{P}\,\mathbb{R} \\
\hline
\forall\, a : \mathbb{R} \bullet (\!|\, a = \{\, x : \mathbb{R} \mid a < x \,\} \\
\forall\, b : \mathbb{R} \bullet b \,|\!) = \{\, x : \mathbb{R} \mid x < b \,\} \\
\forall\, a, b : \mathbb{R} \bullet (\!|\, a \,,\, b \,|\!) = \{\, x : \mathbb{R} \mid a < x < b \,\}
\end{array}
$$

This design is a step in the right direction, but it doesn't prevent the writer from abusing the notation. For example, the following paragraph looks odd but makes perfect sense to *f*UZZ.

$\forall\, a, b : \mathbb{R} \bullet (\!|\, a \,,\, b \,|\!) = b \,|\!) \,,\, (\!|\, a$

When we are asked to interpret (a, b) as an interval, we are, in a sense, being asked to parse a sentence in a new mathematical mini-language that defines intervals. If we want to enforce the usual syntax for intervals, we can introduce new types to represent fragments of the expression so that the operators can only be applied to fragments in some prescribed order.

To illustrate this point, let's expand the example to include closed intervals $[a, b]$ as well as semi-closed intervals $(a, b]$ and $[a, b)$. Regard $(a$ and $[a$ as open and closed right half lines, and $b)$ and $b]$ as open and closed left half lines. Only allow a right half line to be combined with a left half line.

9.3. **Right Half Lines.** Let *RightHalfLine* denote the type of syntactic fragments that define open and closed right half-lines.

$$\mathit{RightHalfLine} ::= (\!|\langle\!\langle \mathbb{R} \rangle\!\rangle \mid [\langle\!\langle \mathbb{R} \rangle\!\rangle$$

Example.

$(\!|\, 0 \in \mathit{RightHalfLine}$

$[\, 1 \in \mathit{RightHalfLine}$

Let $\mathit{rightSet}(R)$ denote the set of real numbers that the right half-line syntactic fragment R represents.

$$
\begin{array}{l}
\mathit{rightSet} : \mathit{RightHalfLine} \longrightarrow \mathbb{P}\,\mathbb{R} \\
\hline
\forall\, a : \mathbb{R} \bullet \mathit{rightSet}((\!|\, a) = \{\, x : \mathbb{R} \mid a < x \,\} \\
\forall\, a : \mathbb{R} \bullet \mathit{rightSet}([\, a) = \{\, x : \mathbb{R} \mid a \leq x \,\}
\end{array}
$$

For example

$0 \notin rightSet((0)$

$1 \in rightSet([1)$

9.4. **Left Half Lines.** Let *LeftHalfLine* denote the type of syntactic fragments that define open and closed left half-lines.

$$LeftHalfLine ::= (\_))\langle\langle\mathbb{R}\rangle\rangle \mid (\_])\langle\langle\mathbb{R}\rangle\rangle$$

For example

$0) \in LeftHalfLine$

$1] \in LeftHalfLine$

Let $leftSet(L)$ denote the set of real numbers that the left half-line syntactic fragment L represents.

$$leftSet : LeftHalfLine \longrightarrow \mathbb{P}\,\mathbb{R}$$

$$\forall\, b : \mathbb{R} \bullet leftSet(b)) = \{\, x : \mathbb{R} \mid x < b \,\}$$

$$\forall\, b : \mathbb{R} \bullet leftSet(b]) = \{\, x : \mathbb{R} \mid x \leq b \,\}$$

For example

$0 \notin leftSet(0))$

$1 \in leftSet(1])$

9.5. **Intersection of Half Lines.** Let $R\,,\,L$ denote the set of real numbers in the intersection of the sets represented by the half-line syntactic fragments R and L.

$$\_\,,\,\_ : RightHalfLine \times LeftHalfLine \longrightarrow \mathbb{P}\,\mathbb{R}$$

$$\forall\, R : RightHalfLine;\ L : LeftHalfLine \bullet$$
$$R\,,\,L = rightSet\ R \cap leftSet\ L$$

For example

$0 \in [0\,,\,1)$

9.6. **Improvements.** After reading the above, my reaction is that it would be clearer to first define the relevant objects using meaningful names and then define the symbols as abbreviations for them.

TODO:

- Fix up the above discussion.
- Use the example environment for examples.
- Improve the Summary. It should summarize the whole article, not just how to simulate mini-languages.

9.7. **Summary.** A lot of standard mathematical notation can be written directly using the built-in capabilities of Z Notation and *f*UZZ. However, mathematics contains many specialized notations including those for intervals, probability, and quantum mechanics (Dirac bra-ket notation). One can view these specialized notations as mathematical mini-languages. As demonstrated above for the case of intervals, it is possible to simulate these notations in Z by introducing new syntactic types along with infix and postfix operators that construct, combine, and reduce fragments of this syntax. The general pattern is for the opening and closing symbols to correspond to prefix and postfix operators that construct fragments, and for internal symbols to correspond to infix operators that combine and then finally reduce fragments. Reduction occurs when the final infix operator is applied and maps the completed mini-language sentence to a some usual mathematical type.

References

[1] Michael Artin et al. "Alexandre Grothendieck 1928-2014, Part 1". In: *Notices of the AMS* 63.3 (Mar. 2016), pp. 242–265. URL: https://www.ams.org/publications/journals/notices/201603/rnoti-p242.pdf.

[2] Henri Poincaré. In: *Science and Method*. Trans. by Francis Maitland. Digitized by the Internet Archive in 2008 with funding from Microsoft Corporation. Thomas Nelson and Sons, London, 1914. Chap. I.II. The Future of Mathematics, p. 34. URL: http://www-history.mcs.st-andrews.ac.uk/Extras/Poincare_Future.html.

[3] Mike Spivey. *The f*UZZ *Manual*. Second Edition. The Spivey Partnership, 2000. URL: https://github.com/Spivoxity/fuzz/blob/59313f201af2d536f5381e65741ee6d98db54a70/doc/fuzzman-pub.pdf.

[4] Cédric Villani. *Cedric Villani: The Hidden Beauty of Mathematics - Spring 2017 Wall Exchange*. 2017. URL: https://youtu.be/v-d0ruh0CHU?t=1823.

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