Notes on $\mathcal{Z}$ for ECS160*

Premkumar Devanbu

March 13, 2001

1 Introduction

$\mathcal{Z}$ is a language that can be used for formal specifications. It is used to deal with systems that have states, inputs and outputs You can informally think of it this way:

$$system.state = f\_s(system.input) \ and$$

$$system.output = f\_o(system.input, system.state)$$

$\mathcal{Z}$ can be used in such situations to describe the behaviour of the system. $\mathcal{Z}$ has been widely accepted in industry, and has been used in a range of different systems, ranging from transportation to IT (Transaction processing systems - such as IBM's CICS) and floating point arithmetic specification for micro computers. $\mathcal{Z}$ offers all the standard advantages of formal specifications:

1. $\mathcal{Z}$ specifications provide a precise statement of what the system should do; the language is designed to make this convenient, easy to understand and express.

2. Since it is a formal language, that can be automatically processed, it can be checked for correctness, consistency, desired properties etc.

3. Certain kinds of partial completeness can be checked; full completeness checking is impossible, since we are modeling the real world (how can we ever be sure we modeled *everything* in the real world that is relevant?)

*I've checked this carefully for mistakes. But if you find any, please let me know as soon as possible!! Thank you.

4. Doing formal specifications early in the software development process can help find faults early, and thus save money.

5. Formal specifications can be used to validate continuously throughout the lifecycle.

Of course, there are some disadvantages:

1. Customers may not be able understand such specifications.

2. There is a fairly level of skill in mathematics that is required.

3. There are some tools but a powerful, integrated environment is lacking (like say, visual C++ or Symantec Cafe for Java (trademarks both).

2 The Basics

$\mathcal{Z}$ specifications clearly state *what* the operations in a system or subsystem *do*, without stating *how* they are implemented. For example, a student in class asked if a "=" in a particular $\mathcal{Z}$ schema was an assignment or a conditional. There are no assignments in $\mathcal{Z}$; it is not a programming language. There are no conditionals either; "conditional" only make sense if you were going to "do" something in either case. $\mathcal{Z}$ specifications just *specify*; they just say what will be *true* about the system, rather than saying *how they should be implemented*. This is done abstractly, by talking the system's behaviour in terms of sets, relations, types, and other mathematical objects. In this sense, $\mathcal{Z}$ is very similar to set theory; in fact, the language of $\mathcal{Z}$ is a logical language.

But, while $\mathcal{Z}$ specifications are written a logical language, it uses symbols that strongly typed, like the language Java. In Java, you cannot say

```
int i,j;
String x;
j=i + x.
```

Likewise, in $\mathcal{Z}$ you cannot say

$$aProf : Professor$$

meaning that *aProf* is of the type *Professor* and then say, later on.

$$primeNumber(aProf)$$

2.1 So what are the types in $\mathcal{Z}$?

$\mathcal{Z}$ has some built-in types, just like in most programming languages. So $\mathcal{Z}$ has type $\mathbb{N}$, which are the natural numbers, and the type $\mathbb{Z}$, which is the set of integers. In addition, $\mathcal{Z}$ also has enumerated types, which are called *free types*. So we can say this:

$$DayofWeek ::= mon \mid tues \mid wed \mid thurs \mid fri \mid sat \mid sun$$

$$Transaction ::= insert \mid delete \mid modify$$

In addition, $\mathcal{Z}$ also has *basic* types. What are these? These are types that would correspond to structures, records or other types of constructed types in languages like C, Pascal and Java. But here in $\mathcal{Z}$, we don't care about the actual implementation, so we only use the *names* of the structures, without stating their actual implementation. So we might say

$$[Student][Book][ClassRoster][Professor][AirlineReservation]$$

to refer to some basic types, which are to be implemented as some non-trivial datastructures. $\mathcal{Z}$ also has power set types, so

$$\mathbb{P}\text{Transaction}$$

refers to the powerset of the $Transaction$ free type above, and likewise

$$\mathbb{P}\text{DayofWeek}$$

2.2 Declarations

All variables used in $\mathcal{Z}$ must be declared. So, we can have declarations such as:

$$thisProf : Professor, aDay : DayofWeek,$$

We can declare variables like this:

$$classSize : \mathbb{N}$$

$$quizGrade : \mathbb{Z}$$

2.3 Some $\mathcal{Z}$ Expression

$\mathcal{Z}$ has the typical expressions from set theory. Now consider the following declarations

$$i, j : \mathbb{N};\ b : Book;\ a : Professor$$

$$onLoan : \mathbb{P}\text{Book};\ aProf : Professor;\ publishedResearchers : \mathbb{P}\text{Professor}$$

Are the following expressions are well typed:

i	$\{onLoan\} \cup \{b\}$
$i * j$	$i + j$
$\{i, j\}$	$\{a, b\}$
$i = j$	$onLoan \subseteq Book$
$i \in Book$	$b \in onLoan$
$\{i, b\}$	$onLoan \cup publishedResearchers$

3 What can you say with these Types?

$\mathcal{Z}$ describes systems by describing the following:

State which describes the states of the system, including all the relevant properties that need to be modeled;

Events which can change various aspects of the states of the system, and

Observations which are ways in some variable with is a part of the state can be read.

3.1 Basic Set Notation

These things are described using formulae that come out of set theory. So first, you get some notation:

$\mathbb{N}$	Natural Numbers
$\mathbb{Z}$	Integers
$a \in S$	a is an element of S

$a \notin S$	a is not an element of S
$\subseteq, \supseteq,$	
$\{\}$,$\phi,$	
$\bigcup, \bigcap$	
$A \setminus B$	Those elements of A that are not in B
$\mathbb{P}$A	powerset of A
$\#Students$	Number of elements in the set $Students$

And there are some logical operators

$\vee, \wedge, \neg$	
$\{D \mid P \bullet x\}$	set of elements x according to declaration such that P
$a \Rightarrow b$	b is true whenever a is true
$a \Leftrightarrow b$	b is true exactly when a is true

And there are some expressions denoting relationships

$x_1 \leftrightarrow x_2$	relationship between sets x_1 and x_2
$x_1 \rightarrow x_2$	total function from x_1 to x_2
$x_1 \nrightarrow x_2$	partial function from x_1 to x_2

Using these notational devices, we can state various things as predicates:

$$waitList, enRolled : \mathbb{P}\text{Student}$$

$$maxClassSize : \mathbb{N}$$

These declare the variables $maxClassSize$ etc. In $\mathcal{Z}$, declared variables represent the *state* of the system. $\mathcal{Z}$ describes the effects of actions by talking about states before and after actions. For this purpose, you can talk about the state of a variable $after$ an action by using the prime " ' " notation; thus $enrolled'$ refers to the set of enrolled students after (some) action.

With these declarations, we can now state *predicates* such as:

$$\#enRolled \leq maxClassSize \ \wedge \ \#enRolled' \leq maxClassSize$$

as an *invariant* which is always true. We can also describe the effect of an operations such as $enroll(s?)$ by

$$waitList' = waitlist \setminus \{s?\} \ , enRolled' = enRolled \bigcup \{s?\}$$

3.2 States

States in $\mathcal{Z}$ are described by *schemas*. Schemas are a combination of a type declaration, and a property specifications. In $\mathcal{Z}$ notation a schema looks like this:

$$
\begin{array}{|l}
\hline
aSchemaName \\
\hline
Declarations \text{ (}or\text{ signatures)} \\
\hline
Predicates \\
\hline
\end{array}
$$

For example,

$$
\begin{array}{|l}
\hline
Class \\
\hline
enrolled : \mathbb{P}\text{Student} \\
\hline
\#enrolled \leq maxClassSize \\
\hline
\end{array}
$$

It may be helpful to think of schemas by analogy to classes in C++. The declarations are like private member variable declarations; the predicates make use of these variables to make assertions. Schema names can refer to other schema names in the declaration part, as a way of "importing" declarations into themselves, and can then make assertions about them. An initial state is described thus:

$$
\begin{array}{|l}
\hline
Initial \\
\hline
Class \\
\hline
enrolled = \phi \\
\hline
\end{array}
$$

In $\mathcal{Z}$ states after operations are shown with a prime "′". Thus, as $Class$ goes through various operations, it's "post-operative" state can be described as:

$$
\begin{array}{|l}
\hline
Class' \\
\hline
enrolled' : \mathbb{P}\text{Student} \\
\hline
\#enrolled' \leq maxClassSize \\
\hline
\end{array}
$$

3.3 Events

Now we can start describing events (or operations). For every state described as a schema, we can have a change Δ operator, defined as follows:

$\Delta Class$

$Class$
$Class'$

or, we can write this full detail by combining $Class$ and $Class'$

$\Delta Class$

$enrolled : \mathbb{P}\text{Student}$
$enrolled' : \mathbb{P}\text{Student}$

$\#enrolled \leq maxClassSize$
$\#enrolled' \leq maxClassSize$

Thus, we can now describe the operation of adding a student to a class:

$AddClass_0$

$\Delta Class$
$s? : Student$

$\#enrolled < maxClassSize$
$enrolled' = enrolled \bigcup \{s?\}$

LIkewise, we can have $DropClass_0$:

$DropClass_0$

$\Delta Class$
$s? : Student$

$s? \in enrolled$
$enrolled' = enrolled \setminus \{s?\}$

3.4 Observations

$\mathcal{Z}$ also has a facility for observing schemas, i.e., checking out the values. Thus for the *Class* schema, we can define an observation schema:

$$
\begin{array}{|l}
\hline
\Xi Class \\
\hline
enrolled : \mathbb{P}\text{Student} \\
enrolled' : \mathbb{P}\text{Student} \\
\hline
\#enrolled \leq maxClassSize \\
\#enrolled' \leq maxClassSize \\
enrolled = enrolled' \\
\hline
\end{array}
$$

Now we can "import" this schema into other observations which have some interesting properties?

$$
\begin{array}{|l}
\hline
ClassSize \\
\hline
\Xi Class \\
numberInClass! : \mathbb{N} \\
\hline
numberInClass! = \#enrolled \\
\hline
\end{array}
$$

Here's another one:

$$
\begin{array}{|l}
\hline
StudentInClass \\
\hline
\Xi Class \\
response! : yes \mid no \\
s? : Student \\
\hline
(s? \in enrolled \land response! = yes) \\
\lor \\
(s? \notin enrolled \land response! = no) \\
\hline
\end{array}
$$

3.5 Exceptions

We can handle error conditions in this way:

$$Message ::= ok \mid alreadyInClass \mid Full \mid notInClass \mid$$

$$
\begin{array}{|l}
\hline
\textit{okMessage} \\
\textit{output}! : \textit{Message} \\
\hline
\textit{output}! = \textit{ok} \\
\hline
\end{array}
$$

One possible error schema:

$$
\begin{array}{|l}
\hline
\textit{AddClassError} \\
\Xi \textit{Class} \\
s? : \textit{Student} \\
\textit{output}! : \textit{Message} \\
\hline
(s? \notin \textit{enrolled} \land \#\textit{enrolled} = \textit{maxClassSize} \land \textit{output}! = \textit{Full}) \\
\lor (s? \in \textit{enrolled} \land \textit{output}! = \textit{alreadyInClass}) \\
\hline
\end{array}
$$

So the complete AddClass:

$$\textit{AddClass} \hat{=} (\textit{AddClass}_0 \land \textit{okMessage}) \lor \textit{AddClassError}$$

Likewise DropClass:

$$
\begin{array}{|l}
\hline
\textit{DropClassError} \\
\Xi \textit{Class} \\
s? : \textit{Student} \\
\textit{output}! : \textit{Message} \\
\hline
(s? \notin \textit{enrolled} \land \textit{output}! = \textit{notInClass}) \\
\hline
\end{array}
$$

and the full DropClass:

$$\textit{DropClass} \hat{=} (\textit{DropClass}_0 \land \textit{okMessage}) \lor \textit{DropClassError}$$

4 Relations and Functions

So far we've been talking about sets such as *Class* and *Student*, and individual members of such sets. $\mathcal{Z}$ has special notations for talking about *pairs* of things. FOr example, consider the following sets.

$$[\textit{Student}][\textit{Class}][\textit{Faculty}][\textit{GradStudent}][\textit{ClassRoom}]$$

Now in $\mathcal{Z}$, you can define a cross product like this:

$$Student \times Class$$

and thus set of all possible subsets of this relation:

$$\mathbb{P}(\text{Student} \times \text{Class})$$

This is also denoted in $\mathcal{Z}$

$$Student \leftrightarrow Class$$

and thus we can declare the *Enrollment* relation as:

$$Enrollment : Student \leftrightarrow Class$$

$\mathcal{Z}$ also has total functions which are denoted by $\rightarrow$, and partial functions, which are denoted by $\nrightarrow$. These can be used to denote the $Instructs$ function (every course has one specific instructor, and $Venue$ partial function (some courses may not meet)

$$Instructs : Class \rightarrow Faculty, \quad Venue : Class \nrightarrow ClassRoom$$

Additionally, in relations, you can have a particular instance of a relation, called a *maplet*. Here's an example maplet of type $Instructs$:

$$ECS160 \mapsto prem$$

(*oops*, is above right?) $\mathcal{Z}$ has notation for updating relations with maplets, for example, to change the instructor for ECS160, I can do the following:

$$Instructs \oplus \{ECS160 \mapsto BillClinton\}$$

The $Instruct$ behaves exactly the same as before for every $Class$, except $ECS160$, which is now maps to $BillClinton$. You can also specify the domain and range of functions (or relations), which corresponds to

$$\mathsf{dom}\ Instructs, \ \mathsf{ran}\ Venue$$

What can we say about the domain of $Instructs$ and the Range of $Venue$ Now, let us define a simple transaction processing system for accounts payable.

Accounts Payable Example First we have two types:

$$[Firms][Dollars]$$

Here's the basic schema.

$$\begin{array}{|l} \textit{AccountsPayable} \\ \hline suppliers : \mathbb{P}\text{Firms} \\ amountDue : Firms \pfun Dollars \\ \hline \mathsf{dom}\ amountDue \subseteq suppliers \end{array}$$

This says that only people you owe money to are your suppliers. We'll see that you can only purchase things from authorized suppliers. *amountsDue* is a partial function; it's a partial function from $Firms$ to $Dollars$. Why is this? becuase we can only owe money to firms who are suppliers, and not all firms are suppliers. However, once a firm gets into our list of suppliers, then each of those must have an amount that we owe to that firm. In the initial state below, we have 3 firms, and we owe no money to each of them.

$$\begin{array}{|l} \textit{Initial} \\ \hline \textit{AccountsPayable} \\ \hline amountsDue = \{Akai \mapsto 0, Midori \mapsto 0, Kinka \mapsto 0\} \\ suppliers = \{Akai, Midori, Kinka\} \end{array}$$

(Ooops, is the specification the dom of *amountDue* correct? Should it be a $\subseteq$ of *suppliers*, or equal to it?) Now we consider a purchase transaction:

$$\begin{array}{|l} \textit{PurchaseTransaction} \\ \hline \Delta \textit{AccountsPayable} \\ supplier? : Firm \\ amount? : Dollar \\ \hline supplier? \in suppliers \\ amountsDue' = amountsDue \oplus \\ \qquad \{supplier? \mapsto ((amountsDue\ supplier?) + amount?)\} \end{array}$$

and a payment transaction

$$
\begin{array}{|l}
\hline
\textit{PaymentTransaction} \\
\Delta AccountsPayable \\
supplier? : Firm \\
amount? : Dollar \\
\hline
supplier? \in suppliers \\
amountsDue' = amountsDue \oplus \\
\qquad \{supplier? \mapsto ((amountsDue\ supplier?) - payment?)\} \\
\hline
\end{array}
$$

We can also have a way to add suppliers

$$
\begin{array}{|l}
\hline
\textit{AddSupplier} \\
\Delta AccountsPayable \\
supplier? : Firm \\
\\
\hline
amountDue' = amountDue \bigcup \{supplier? \mapsto 0\} \\
suppliers' = suppliers \bigcup \{supplier?\} \\
\\
\hline
\end{array}
$$

And an observation about how much money we owe anybody?

$$
\begin{array}{|l}
\hline
\textit{WhatDue} \\
\Xi AccountsPayable \\
supplier? : Firm \\
amount! : Dollars \\
\hline
supplier? \in suppliers \\
amount! = (amountDue\ \ supplier?) \\
\hline
\end{array}
$$

5 Sequences, and some datastructures

$\mathcal{Z}$ can be used to model abstract datastructures (Stacks of anything, Double-ended queues of anything, binary trees of anything that has a total order,

etc). We'll look at stacks. First, what is a sequence in $\mathcal{Z}$? A sequence is declared thus:

$$x : \mathsf{seq}\ T$$

where T is any of types in $\mathcal{Z}$. This is actually equivalent to the declaration:

$$x : \mathbb{N}_1 \pfun T,\ where\ \ \mathsf{dom}\ x = 1 \ldots \#x$$

where "$\# x$ is the length of the sequence. $\mathbb{N}_1$ refers to the natural numbers ≥ 1. Sequences are shown within angle brackets, or as relations:

$$< 9, 14, 23, 33 >\ \ or\ as$$

$$\{1 \mapsto 9, 2 \mapsto 14, 3 \mapsto 23, 4 \mapsto 33\}$$

and Empty sequences are shown as

$$<>$$

Sequences support several operations:

selection

$$< mon, tues, wed, thurs, fri, sat, sun > 4 = thurs$$

concatenation

$$< mon, tues > \frown < wed, thurs >=< mon, tues, wed, thurs >$$

head

$$\mathrm{head} < mon, tues, wed >= mon$$

last

$$\mathrm{last} < mon, tues, wed >= wed$$

tail

$$\mathrm{tail} < mon, tues, wed >=< tues, wed >$$

front

$$\mathrm{front} < mon, tues, wed >=< mon, tues >$$

5.1 Stack Example

Stack[*T*]

$st : seqT$

StackInit

Stack[*T*]

$st = <>$

push[*T*]

$\Delta stack$
$item? : T$

$st' = < item? > \frown st$

pop[*T*]

$\Delta stack$
$item! : T$

$item! = \text{head } st$
$st' = \text{tail } st$

What's missing? errors, of course. How would I describe stack underflow? How can describe the size of the stack etc? (Hint: add another variable of type $\mathbb{N}$ to the stack)

5.2 A Sequential File

Such a file is a Sequence of records stored one after another on an external storage device. A file can be either an input or an output file, but not both at the same time. In a sequential file, access to the nth record is only possible if $n-1$ records are first read in order. T is the type of the record (an unspecified

basic type)

$$
\begin{array}{|l}
\hline
\textit{seqFile}[T] \\
\hline
file : \ \mathsf{seq}\ T \\
unread : \ \mathsf{seq}\ T \\
alreadyread : \ \mathsf{seq}\ T \\
mode : FileMode \\
\hline
alreadyread^\frown unread = file \\
\hline
\end{array}
$$

$$FileMode ::= input \mid output$$

Keep in mind here that read and write are incompatible. In this example, a File cannot be open for read and write at the same time. That simplifies matters for the purposes of this example. Now let's open a file for input:

$$
\begin{array}{|l}
\hline
\textit{openRead}[T] \\
\hline
\Delta seqFile[T] \\
\hline
mode' = input \\
unread' = file \\
file' = file \\
\hline
\end{array}
$$

why is unread' = file? and file' the same as file? Now let's do a read:

$$
\begin{array}{|l}
\hline
\textit{Read}[T] \\
\hline
\Delta seqFile[T] \\
x! : T \\
\hline
mode = input \\
unread \neq <> \\
< x! > ^\frown unread' = unread \\
alreadyread' = alreadyread^\frown < x! > \\
mode' = mode \\
file' = file \\
\\
\hline
\end{array}
$$

Let's open a file for output:

$$
\begin{array}{|l}
\hline
openWrite[T] \\
\Delta seqFile \\
\hline
mode' = output \\
file' = file \\
\\
\hline
\end{array}
$$

Now let's write to this file:

$$
\begin{array}{|l}
\hline
Write[T] \\
\Delta seqFile[T] \\
x? : T \\
\hline
mode = output \\
file' = file^\frown < x? > \\
mode' = mode \\
\hline
\end{array}
$$

We can also test of end of file?

$$Boolean ::= yes \mid no$$

$$
\begin{array}{|l}
\hline
eof[T] \\
\Xi seqFile[T] \\
answer! : Boolean \\
\hline
((unread = <>) \land (answer! = yes)) \\
\lor \\
((unread \neq <>) \land (answer! = no)) \\
\hline
\end{array}
$$

6 Finally...

Now you should be able to specify other datastructures in $\mathcal{Z}$. Try Queues. Stack is last in first out, queue is first in, first out. So you should do enque (add item to queue) deque (remove item) as events, and length of queue as an observation. Plus consider error conditions.