Towards a Formal Semantics for the Z Notation

by

Mike Spivey

Technical Monograph PRG-41
October 1984

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

Contents.

1. Introduction.

The Z notation [Sufrin 84, Morgan 84] is a language for expressing formal specifications of computing systems. It is based on a typed set theory, and the notion of a "schema" is one of its central features. A schema consists of a collection of named objects specified by some axioms, and Z provides a system of notation for combining schemas which conveniently allows large specifications to be built up in stages.

The first part of this paper surveys approaches to the semantics of programming languages and their relevance to specification languages, then gives a sketch of a denotational semantics for Z based on the notion of a "variety". In this semantics, the meaning of a specification is taken to be the collection of all its models. The second part lists some of the benefits resulting from a study of the formal semantics of specification languages. An appendix describes the view of set theory which underlies the semantics.

This paper is based on a talk given at the Programming Research Group in May, 1984. I should like to thank Bernard Sufrin and Kavi Arya for their helpful comments. The financial support of the Science and Engineering Research Council of Great Britain is acknowledged.

2. Towards a Formal Semantics.

This section begins with a brief look at the approaches which have been taken to describing the semantics of programming languages and an assessment of their relevance to the semantics of specification languages. One of these techniques, denotational semantics, is then applied to the semantics of Z: we first show how suitable semantic domains can be constructed using the notion of a "variety", then as examples show how some operations on schemas can be described in the model.

2.1. Programming Languages and Specification Languages.

Most work on formal semantics by computer scientists has dealt with programming languages, but many of the techniques can be applied to specification languages with a few modifications. Approaches to the the formal semantics of programming languages can be classified into three broad groups:

Denotational Semantics [Stoy 77]. This starts with some semantic domains - spaces of "abstract meanings" - on which operations are defined corresponding to the constructs of the language. The structure of programs is reflected by "abstract syntax", and this is related to the semantic space by semantic equations. The meaning of each class of syntactic objects - expressions, statements and so on - is given by defining a semantic function to map program fragments onto their abstract meanings.

For programming languages, appropriate semantic domains may be constructed using the theory of Scott domains and approximable mappings [Scott 76, 81, 82]. For specification languages, the semantic space may be taken as either the world of "theories" [Burstall & Goguen 80, 82] - the meaning of a specification being

the collection of all the propositions it entails - or the world of "varieties" - the meaning of a specification being the collection of all its models. We shall see later how the Z notation may be given a semantics based on varieties.

Axiomatic semantics. Here, the semantics of a programming language is defined by giving rules for deducing properties of programs. An example of this approach is the well-known "partial-correctness formulae" due to Tony Hoare [Hoare 69]: the formula P { S } Q is valid exactly if program S, when started in a state satisfying predicate P, cannot terminate in a state which fails to satisfy the predicate Q. The semantics of a programming language can be characterized - well enough at least for program development - by giving a set of rules for deducing correctness formulae.

For specification languages, an axiomatic semantics might involve rules for reasoning about specifications. Such rules might be used for proving that one specification refines another, or for deducing consequences of design decisions.

Operational semantics. The semantics of a programming language may also be given by describing an abstract machine which enacts the computation encoded by a program. This approach is the foundation for the Vienna Definition Language [Lucas & Walk 69], which was used to give a formal semantics for PL/I. An operational definition of a programming language is particularly well-suited to proving the correctness of implementations of the language, but does little to facilitate reasoning about programs.

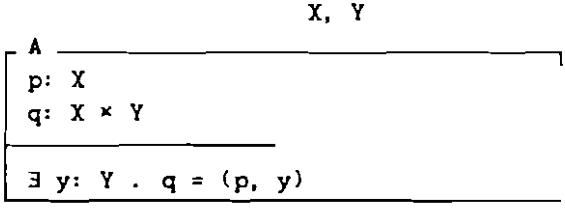
Some specification languages can be regarded as very high-level programming languages, and these have a natural operational semantics - indeed, in logic programming [Kowalski 79] one tries to give an operational semantics to the predicate calculus itself. Specifications written in such specification languages may be executed by a machine and thus used to create "rapid prototypes".

Not all specification languages can be executed, however. In particular, some can describe functions which are not effectively computable, and an operational semantics for these can be at best partial: the Z notation is such a specification language. It might perhaps be argued that specifications are "for reasoning about" in the same sense that programs are "for executing", and that in consequence the natural counterpart for specification languages of the operational semantics of programming languages is to be found in axiomatic semantics.

The rest of this section introduces a semantic theory which can be used to give a denotational semantics to the Z notation.

2.2. Signatures.

Consider the following example of a simple schema:



This contains some declarative information above the horizontal dividing line, and some further information conveyed by the axioms below it. The declarative part of the schema introduces the following information:

- The given-set names X and Y.
- The variable names p and q.
- Some type information: p has type X, and q has type X × Y.

This information forms the signature of the schema A. Formally, the notion of signature is defined as follows:

```
SIG
given: ℙ GNAME
vars: ℙ VNAME
type: VNAME ⇸ TYPE
---
type ∈ (vars → Type(given))
```

A signature contains an alphabet of given-set names, drawn from the set GNAME, an alphabet of variable names, drawn from the set VNAME, and a function which assigns a type to each variable name. The types must be formed from given-set names in the alphabet: the function Type yields the set of such types given the alphabet. The signature of the schema A is

```
μ SIG |
    given = { "X", "Y" }
    vars = { "p", "q" }
    type = { "p" ↦ "X", "q" ↦ "X × Y" }.
```

There are several advantages of separating this declarative information from the information conveyed by the axioms:

- The signature of a schema may be regarded as an interface by means of which it may be assembled with other schemas to form a larger specification.

- The theory of signatures is decidable. This means that the well-formedness of specifications in terms of the rules for signatures is particularly suited to mechanical checking.

- The type information contained in signatures has a pragmatic value in preventing errors in specifications.

2.3. Varieties: Meanings for Schemas.

As well as this declarative information, the schema A also contains information conveyed by the axiom below the horizontal line. This information can be captured by regarding A as characterizing a class of "structures". For example, the structure

$$\begin{aligned} &"X" \mapsto \mathbb{N} \\ &"Y" \mapsto \{ a, b, c \} \\ \\ &"p" \mapsto 3 \\ &"q" \mapsto (3, b) \end{aligned}$$

satisfies the axiom, but the structure

$$\begin{aligned} &"X" \mapsto \{ f, g, h \} \\ &"Y" \mapsto \{ a, b, c \} \\ \\ &"p" \mapsto h \\ &"q" \mapsto (g, b), \end{aligned}$$

although it also accords with the signature, fails to satisfy the axiom, because the value of p is not the same as the first component of the value of q.

A structure takes certain given-set names and variables and gives them values in the "world" of sets. If we assume a set W which models a universe of sets, and a binary relation $\in$ which models the membership relation between sets, we can characterize structures as pairs of functions from GNAME and VNAME into W:

$$\begin{array}{|l} \text{STRUCT} \\ \hline gset : GNAME \pfun W \\ val : VNAME \pfun W \\ \hline \end{array}$$

The universe W can be described formally by giving the axioms of set theory as a specification: further details are given in the appendix.

The first requirement on the structures for a schema is that they be consistent with the signature of the schema: the domains of the gset and val mappings should be the alphabets of the signature, and the value given to each variable must be in the "carrier" for the corresponding type. We define the function Struct to give the set of structures consistent with a signature:

```
Struct: SIG → ℙ STRUCT
```

```
Struct =
    λ SIG .
        { STRUCT |
            dom gset = given
            dom val = vars
            ∀ v: vars .
                val v ∈ Carrier gset (type v) }
```

The function Carrier gives the set of elements of a type by interpreting type-constructors as operations in the world of sets.

Now a variety - the meaning of a schema - can be defined as a signature together with a set of structures for the signature:

```
VARIETY
sig: SIG
models: ℙ STRUCT
---
models ⊆ Struct(sig)
```

The set models will typically be smaller than Struct(sig) because some structures will fail to satisfy the axioms of the schema. For example, the schema A characterizes a variety with the signature given above. The models component is

```
models =
    { STRUCT |
        dom gset = { "X", "Y" }
        dom val = { "p", "q" }
        val "p" ∈ gset "X"
        val "q" ∈ product(gset "X", gset "Y")
        ∃ yy: W | yy ∈ gset "Y" .
            val "q" = opair(val "p", yy) }
```

Here product denotes the Cartesian product operation in the world of sets, and opair denotes the operation of ordered-pair formation. To be a model for A, a structure must give values to the given-set names X and Y and the variables p and q. The value of p must be in X, and the value of q must be in X × Y. Finally, the axiom of A must be satisfied.

2.4. Some operations on schemas.

These concepts can be used to build semantic domains for a denotational semantics of the Z notation. As examples, we give the semantics of some of the operations of the schema-calculus. These operations may be used to build up "schema-expressions"; we show the operations of schema-conjunction and disjunction, and the projection of one schema on the variables of another:

```
SEXPR ::=
    ... | SEXPR ∧ SEXPR | SEXPR ∨ SEXPR
        | SEXPR ↾ SEXPR | ...
```

Consider the operation of schema-conjunction. Given two schemas A and B, the signature of their conjunction $A \wedge B$ is formed by joining their signatures with identification of common variables: the conjunction can only be formed if the common variables have the same types in A and B. The models of $A \wedge B$ are those which satisfy, in a certain sense, both the axioms of A and the axioms of B.

The meaning of a schema-expression is defined by a semantic function sexpr which, given an environment of schema-definitions, maps schema-expressions to varieties:

$$\mathit{sexpr} : ENV \rightarrow SEXPR \nrightarrow VARIETY$$

$$\ldots$$
$$\mathit{sexpr}\ \rho\ [\![\ S_1 \wedge S_2\ ]\!] == \mathit{combine}(\mathit{sexpr}\ \rho\ [\![\ S_1\ ]\!],\ \mathit{sexpr}\ \rho\ [\![\ S_2\ ]\!])$$
$$\ldots$$

The operation combine puts together the varieties corresponding to the arguments of a schema-conjunction in the way described informally above. The signatures are joined using an auxiliary function join, and the class of models is defined in terms of the function restrict, which recovers a structure for a smaller signature from one for a larger signature in which it is included.

```
combine: VARIETY × VARIETY ⇸ VARIETY
─────────────────────────────────────
combine(θVARIETY₁, θVARIETY₂) ==
    μ VARIETY' |
        sig' == join(sig₁, sig₂)
        models' =
            { M: Struct(sig') |
                restrict sig₁ M ∈ models₁ &
                restrict sig₂ M ∈ models₂ }
```

This definition bears a striking resemblance to the definition of parallel composition in CSP [Hoare 83] if we regard signatures and models as analogous to alphabets and traces respectively:

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q$$

$$traces(P \parallel Q) = \{ t \in \alpha(P \parallel Q)^* \mid t \upharpoonright \alpha P \in traces(P) \ \&\ t \upharpoonright \alpha Q \in traces(Q) \}.$$

The disjunction of two schemas A and B is defined similarly: the signatures are still joined in the same way, but the models are those which satisfy either the axioms of A or those of B:

```
─────────────────────────────────────
...
sexpr ρ [ S₁ ∨ S₂ ] ==
    disjoin(sexpr ρ [ S₁ ], sexpr ρ [ S₂ ])
...
```

The operation disjoin is defined exactly like combine, except the models component of the result is given by

```
models' =
    { M: Struct(sig') |
        restrict sig₁ M ∈ models₁ ∨
        restrict sig₂ M ∈ models₂ }.
```

The semantics of schema-projection is defined of an operation project:

```
...
sexpr ρ [ S₁ ↾ S₂ ] ==
    project(sexpr ρ [ S₁ ], sexpr ρ [ S₂ ])
...
```

The standard definition of A ↾ B hides those components of A which are not in the signature of B. It is required that the signature of A includes the signature of B, and the axioms of B are ignored. The models in the result are those structures for the signature of B which can be obtained by restriction from a model of A.

```
project: VARIETY × VARIETY ⇸ VARIETY

project(θVARIETY₁, θVARIETY₂) ==
    μ VARIETY' |
        sig₂ subsig sig₁
        sig' = sig₂
        models' = (restrict sig₂)( models₁ )
```

An alternative definition might require the models in the result to be models for B as well: this effect would be obtained by setting

$$models' = models_2 \cap (\mathrm{restrict}\ sig_2)(models_1).$$

These examples show how the theory of varieties can be applied to the semantics of some simple Z constructs. Of course, for a full semantics of Z it is necessary to describe the environments which record definitions of schemas and the process by which generic schemas can be instantiated with particular types, as well as the mathematical sublanguage in which the axiom parts of schemas are expressed.

The next section goes on to examine some of the benefits which might be expected from a formal semantics of Z.

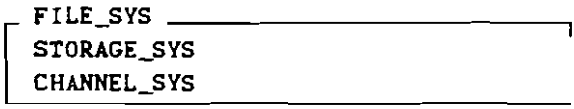
3. The Benefits of a Formal Semantics.

Several benefits can be derived from a study of the formal semantics of the Z notation. Such a study can help in the understanding of the specification process by clarifying certain desirable properties of specifications. It can also help in the design of better specification languages by allowing a critical comparison of specification techniques and notations. Finally, a formal semantics of the specification language is a necessary prerequisite for rigorous development of software tools to assist in the task of specification and program development.

3.1. Understanding the Specification Process.

The formal semantics of Z helps us to understand the issues involved in writing Z specifications. Firstly, the semantics can provide a foundation for a logical calculus for reasoning about specifications. Such a calculus may be regarded as an axiomatic semantics of Z, and can be derived from the denotational semantics in the same way that the soundness of Hoare-style proof rules for a programming language can be proved from the denotational semantics of the language. The semantics of schema-conjunction given above justifies a proof-rule for deriving theorems about schemas built up using the operations from theorems about their component schemas.

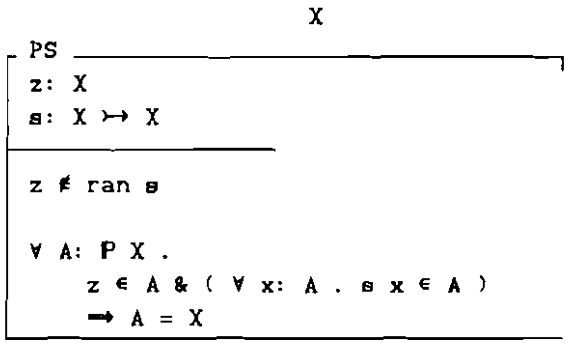
A semantics based on varieties helps to support our intuition about composite specifications better than one based entirely on a textual view of specifications. For example, if we say



we intend that models of the file-system should contain aspects which constitute models of STORAGE_SYS and CHANNEL_SYS, and not merely that a text describing FILE_SYS may be obtained by joining texts describing the two components.

The denotational semantics also allows the rigorous formulation of notions essential to program development. For example, it allows us to say exactly what it means for one specification to refine another, or for a specification to be consistent or complete, and allows techniques to be developed for proving that specifications enjoy these desirable properties. A simple example of the notions of consistency and completeness is the problem of characterizing the natural numbers.

Most specifications use the symbol $\mathbb{N}$ - but what does it mean? We might decide that $\mathbb{N}$ will always denote a particular object in the world of sets: in this case, $\mathbb{N}$ must be "built-in" to the notation. But this is unnecessary, since everything we require of $\mathbb{N}$ may be captured by a small specification of a "Peano system": there must be a set X, together with a "zero" z and a "successor function" s, and these must have certain characteristic properties, including an induction principle.



Now there are two questions to settle:

- Have we really captured all we need to know about ℕ - is the specification complete?
- Is the specification free from contradictions - is it consistent?

We can prove completeness by an inductive argument within the framework of Z - we show that any two models are uniquely isomorphic, that is, there is a unique bijection between them which preserves the zero and successor function:

```
[X, Y]
PS[X]; PS'[Y] ⊢
    ∃! h: X ⤖ Y .
        h z = z' &
        ( ∀ x: X . h (s x) = s' (h x) ).
```

To prove consistency, we must step outside the context of Z and explicitly construct a model of the specification in the world of sets.

These methods can also be applied to establish the consistency and completeness of other specifications, for example those arising from abstract syntax notation:

```
BT ::= leaf ⟨⟨ ℕ ⟩⟩ | node ⟨⟨ BT × BT ⟩⟩
```

means the same as

```
BT -- Binary trees.

leaf: ℕ ↣ BT
node: BT × BT ↣ BT
---------------------------------
ran leaf ∩ ran node = ∅

∀ A: ℙ BT .
    ( ∀ n: ℕ . leaf n ∈ A ) &
    ( ∀ b₁, b₂: A . node(b₁, b₂) ∈ A )
    ⇒ A = BT
```

3.2. Studying Specification Languages.

Formal semantics has proved to be a powerful technique for studying programming languages: it allows the subtle nuances of meaning in, for example, the Algol 60 for statement to be brought out and discussed. The indispensable advantage offered by formal semantics is the possibility of being perfectly precise about the concepts involved, and we can expect this advantage to apply to specification languages just as much as programming languages.

Denotational semantics provides a view of programs and specifications which abstracts away from inessential details of syntax and presentation: in the semantic theory described in section 2, for example, two schemas whose axioms have the same content will be modelled by the same variety, even if the axioms are vastly different in form. This view makes it easy to investigate new language constructs: for example, a simple semantics can be given to the use of a schema as a predicate, and this notion proves quite useful in certain kinds of specification problem. During data refinement of a specification, it is neccessary to show that each operation preserves the invariant relation I between abstract and

concrete states. This requirement can be expressed by the theorem

$$I \wedge AOP \wedge COP \vdash I'$$

where AOP specifies the abstract operation and COP specifies its realisation, and the schema I' on the right-hand side is used as a predicate.

A study of the formal semantics of a notation allows a critical examination of the rigour of the notation -- indeed, such a critical examination is a necessary part of the study. If any construct in the notation is ambiguous or ill-founded, it will be impossible to write down the semantics, and this will encourage a closer examination of the construct in question. An example of this phenomenon in the Z notation is the facility for global generic definitions.

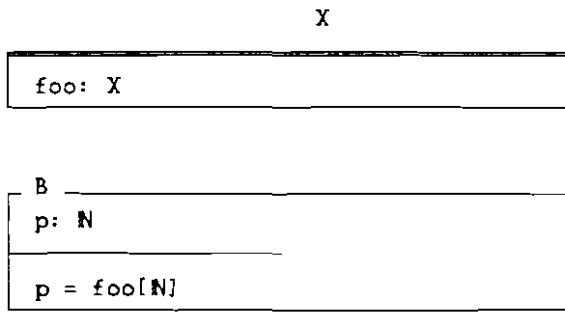
We often want to make definitions generic in some given sets, for example, the concatenation operator on sequences, which is defined for every possible type of element:

$$X$$

$$\_ * \_ : seq[X] \times seq[X] \rightarrow seq[X]$$

$$\ldots$$

It turns out to be difficult to describe this sort of definition using the theory. Whilst such definitions make obvious sense in simple cases, more complex cases seem to cause problems. For example, suppose foo[X] is defined to be a member of the set X, and consider the schema B:



How many models does the schema B have? Is there just one model, corresponding to a single possible value of p? If so, what reason is there to prefer one value over any other?

Or does B have many different models, each with a different value for p? In this case, the composite schema B ∧ B' will have models in which p takes one value and p' takes a different value. This seems to contradict the argument which deduces p = p' from the axioms p = foo[ℕ] and p' = foo[ℕ], together with the transitive property of equality.

3.3. Implementing Support Tools.

Since formal specifications are formal texts, they are amenable to manipulation by machine. Several kinds of software tool can assist in the process of program development: checking programs of various kinds can be used to detect simple errors in specifications, and machine assistance with the proof of theorems about specifications may make formal verification of designs feasible. Many of the proofs needed during the development process are very shallow, but contain a mass of detail: this is just the kind of proof for which mechanical assistance is most valuable.

These software tools, if they are to be effective, must be designed in the same rigorous way as any other program, and this means that the development must start with a formal specification. For the simplest kinds of analysis, a formal syntax for the specification language is all the information needed, but a program to check for violations of the type rules, for example, will require these rules to be formulated rigorously. A mechanised proof assistant will need a collection of inference rules for reasoning about specifications. Whilst these collections of rules might all be put together in an arbitrary way, it is clear that greater confidence can be placed in their consistency if they are all derived from a formal semantics for the specification language.

4. Summary.

The paper began by sketching a semantic theory which can be used as the basis for a semantics of the Z notation. First the notion of a signature was introduced as representing the declarative information in a schema, then the notion of a variety, which consists of a signature together with a class of structures. Varieties can be taken as a model for the meaning of schemas, and this leads to a denotational semantics for Z. As an example, the semantics of schema-conjunction was given by introducing a corresponding operation on varieties.

The second part of the paper surveyed the likely applications arising from a study of the formal semantics of Z. Firstly, a formal semantics can help in understanding the process of writing formal specifications. It can provide rules for reasoning about specifications, and a rigorous formulation of ideas, such as consistency, completeness and refinement, which are essential to program development.

Formal semantics also provides an indispensible tool for studying specification languages. The meaning of new language constructs and the rigour of established ones can both be discussed conveniently at the level of abstraction made possible by denotational semantics.

Finally, a formal semantics for the specification language is a necessary part of the specification of software tools to support program development.

Appendix. The World of Sets.

The formal semantics sketched in section 2 depends on a formulation of the axioms of set theory as a Z specification. This specification begins by introducing a given-set name W to denote the "world" of sets, and a binary relation ∈ on W to denote the membership relation. The first requirement is that membership be extensional, i.e. that any two sets with the same members are equal:

```
W -- World of sets.

_ ∈ _ : W ↔ W

∀ x, y: W .
    ( ∀ z: W . z ∈ x ⇔ z ∈ y ) ⇒ x = y
```

The axioms corresponding to set-theoretic constructions give rise to functions on W: for example, the power-set axiom gives rise to the function power:

```
power: W → W

∀ x, y: W .
    x ∈ power y ⇔ ( ∀ z: W . z ∈ x ⇒ z ∈ y )
```

For the axiom of separation, properties of sets are represented by subsets of W. A subset S of W may be a "proper class" in the sense that there may be no "set" in W whose "members" under ∈ coincide with the members of S. The axiom of separation gives rise to a function `filter`:

$$\text{filter}: W \times \mathbb{P} W \to W$$

$$\forall x, y: W; S: \mathbb{P} W \bullet x \in \text{filter}(y, S) \Leftrightarrow x \in y \wedge x \in S$$

By omitting the axiom of replacement, we obtain a version of set theory with standard models within ordinary Z-F set theory: for example, it is well-known that $V\lambda$, for λ a limit ordinal larger than omega, satisfies all the Z-F axioms except replacement [Enderton 77].

References.

[Burstall & Goguen 80]

Burstall, R.M., & Goguen, J.A., "The Semantics of CLEAR, a Specification Language", Internal Report CSR-65-80, Department of Computer Science, University of Edinburgh, 1980.

[Burstall & Goguen 82]

Burstall, R.M., & Goguen, J.A., "Algebras, Theories and Freeness: An Introduction for Computer Scientists", in "Theoretical Foundations of Programming Methodology", M. Broy & G. Schmidt (Eds.), D. Reidel, 1982.

[Enderton 77]

Enderton, H., "Elements of Set Theory", Academic Press, 1977.

[Hoare 69]

Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", Comm. ACM 12 (1969) pp. 576-580, 583.

[Hoare 83]

Hoare, C.A.R., "Notes on Communicating Sequential Processes", Technical Monograph PRG-33, Programming Research Group, University of Oxford, 1983.

[Kowalski 79]

Kowalski, R., "Logic for Problem Solving", North-Holland, 1979.

[Lucas & Walk 69]

Lucas, P., & Walk, K., "On the Formal Description of PL/I", Annual Review in Automatic Programming, 6 (1969).

[Morgan 84]

Morgan, C.C., "Schemas in Z - A Preliminary Reference Manual", Programming Research Group, University of Oxford, 1984.

[Scott 76]

Scott, D.S., "Data Types as Lattices", SIAM Journal of Computing, 5 (1976) pp. 522-587.

[Scott 81]

Scott, D.S., "Lectures on a Mathematical Theory of Computation", in "Theoretical Foundations of Programming Methodology", M. Broy & G. Schmidt (Eds.), D. Reidel, 1982.

[Scott 82]

Scott, D.S., "Domains for Denotational Semantics", presented at ICALP '82, Aarhus, Denmark, July 1982.

[Stoy 77]

Stoy, J.E., "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", MIT Press, 1977.

[Sufrin 84]

Sufrin, B.A., "Notes for a Z Handbook: Part I - The Mathamatical Language", Programming Research Group, University of Oxford, 1984.