Modularity for Z

ZSRC Document 182*

R.D. Arthan
ICL, Eskdale Road,
Winnersh, Berks.
RG11 5TT
rda@win.icl.co.uk

5 September 1995

1 INTRODUCTION

Z as described in older versions of the draft standard and in the books [3, 4] gives no facility for a specification to be organised into groups of paragraphs which can be separately developed and used. Presentation of a Z specification as a group of documents was supported by various features of the language as described in [2]. Other proposals for modularity for Z have been made from time to time, e.g., [1], and some tools for Z support a modularity feature. However, no modularity scheme has gained popular acceptance.

It has generally been understood during the standardisation work that a modularity feature would be added to the language. Some time ago Colin Parker did some work on the related topic of invocation of the Z toolkit, reported in ZSRC document D-121; however, the scheme in D-121 does not support separate type-checking.

This document proposes a modularity feature for Z which offers a modest, but useful, level of functionality. There are doubtless good arguments for schemes offering much more; the simple proposal made here is offered as a starting point.

In outline, the scheme proposed here simply allows the user to associate names with sequences of paragraphs. In order to avoid conflict with programming language terminology, these named sequences of paragraphs are called "sections". The user may provide a list of "parent" sections for a section. Definitions or axioms available in a parent are available for use in its children.

The remainder of this document is structured as follows:

Section 2	describes the proposed scheme using the same style as the standard.
Section 3	discusses various issues in the design of the scheme.
Section 4	indicates how the Z Toolkit might be organised into sections and how it might be extended to incorporate a theory of real numbers. (N.b., this part of the document does not constitute a specific proposal; it is just an example of how the proposed modularity scheme might be used in the standard itself.)

*Author's reference: DS/FMU/IED/WRK059, Issue 3; Issues 1 and 2 were distributed as ZSRC documents 178 and181; I would like to thank those who commented on the earlier issues, particularly those who attended the Limerick ZSRC meeting.

2 THE PROPOSAL

To define a language construct in the style of the draft standard, one has to give the following:

- Syntax description, comprising:
  - representation syntax (i.e., concrete syntax);
  - abstract syntax;
  - transformations (i.e., the mapping of representation syntax to abstract syntax).
- Semantics description, comprising:
  - type and value environments; (i.e. the semantic domains)
  - type rules ("static" semantics);
  - value rules ("dynamic" semantics).

These items for the proposed modularity feature are described below.

2.1 Syntax

2.1.1 Representation Syntax

The representation syntax for a specification allows either a sequence of sections or a sequence of paragraphs. A section comprises a section header followed by a sequence of paragraphs. A section header uses the keyword 'section' to introduce the section name and may also contain a list of parent section (whose definitions are thereby made available for use in the section).

$$
\begin{array}{lll}
\mathit{Specification} & = & [\mathit{Narrative}],\ \{\mathit{Section}\} \\
 & | & \mathit{Paragraphs};
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Section} & = & '\mathit{section}',\ \mathit{Word}, \\
 & & ['\mathit{parents}',\ \mathit{Word},\ \{',',\ \mathit{Word}\},\ '\bullet'], \\
 & & \mathit{Paragraphs};
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Paragraphs} & = & [\mathit{Paragraph}],\ \{\mathit{Narrative},\ \mathit{Paragraph}\},\ [\mathit{Narrative}];
\end{array}
$$

The above is intended to replace the production for *Specification* in the current draft. Apart from the name, the production for *Paragraphs* above is the same as that given for *Specification* in the current draft.

The second alternative for specification is for backwards compatibility and for use by those who just want to write simple linear specifications which make free use of the standard toolkit. A specification comprising just a sequence of paragraphs is to be transformed as if preceded by *section Specification parents Toolkit* (see below).

2.1.2 Abstract Syntax

The abstract syntax of a specification comprises a sequence of sections. A section is a named sequence of paragraphs together with a list of parent section names.

$$SPEC = SECTION, ..., SECTION$$

$$SECTION = section\ NAME\ parents\ NAME, ..., NAME \bullet PARAGRAPHS$$

$$PARAGRAPHS = PAR, ...\ PAR$$

The above is intended to replace the abstract syntax for *SPEC* in the current draft.

2.1.3 Transformations

We need two extra transformation brackets[1]:

Brackets	Argument
$[(\ \_\ )]^{Pars}$	Paragraphs
$[(\ \_\ )]^{Sec}$	Section

The transformation for *Paragraphs* is just like that for *Specification* in the current draft:

Production	Representation	Abstract
$[Paragraph]$, $\{Narrative, Paragraph\}$, $[Narrative]$	P_1 $Narrative$ ... $Narrative$ P_n	$[(P_1)]^{Par}$, ..., $[(P_n)]^{Par}$

The transformations for *Section* are:

Production	Representation	Abstract
$'section', Word,$ $['parents', Word, \{',', Word\}, '\bullet'],$ $Paragraphs;$	$section\ X$ $Paragraphs$ $section\ X$ $parents\ P_1, ...\ P_n \bullet$ $Paragraphs$	$section\ X\ parents \bullet$ $[(Paragraphs)]^{Pars}$ $section\ X$ $parents\ P_1, ...\ P_n \bullet$ $[(Paragraphs)]^{Pars}$

The transformations for *Specification* are:

Production	Representation	Abstract
$[Narrative]$, $\{Section\}$	$Narrative$ $M_1, ..., M_n$	$[(M_1)]^{Sec}$, ..., $[(M_n)]^{Sec}$
$Paragraphs$	$Pars$	$[(section\ Specification\ parents\ Toolkit\ Pars)]^{Sec}$

[1] Unfortunately, the LaTeX macros used in the standard were not available to the author at the time of writing; I have therefore used ASCII approximations to the various brackets.

2.2 Semantics

2.2.1 Environments

A section type environment is a partial function from section names to pairs comprising a set of names and a type environment as in the current draft standard. The set of names identifies the ancestry of the section in question.

$SectionTenv == Name \mathrel{+\mkern-14mu\rightarrow} (\mathbb{P}\ Name \times TEnv)$

A section value environment is a partial function from section names to value environments:

$SectionEnv == Name \mathrel{+\mkern-14mu\rightarrow} Env$

There are semantic brackets[2] as follows for sections and specifications:

Bracket	Argument	Forms
$<\{ \_ \}>$	Section	$<\{ \_ \}>^{\mathcal{T}}$, $<\{ \_ \}>^{\mathcal{M}}$
$<(\_)>$	Specification	$<(\_)>^{\mathcal{T}}$, $<(\_)>^{\mathcal{M}}$

2.2.2 Type Rules

The type functions for sections and specifications are as shown in the following table:

Name	Form	Sort
Section Type	$<\{ Sec \}>^{\mathcal{T}}$	$SectionTenv \mathrel{+\mkern-14mu\rightarrow} SectionTenv$
Specification Type	$<(Spec)>^{\mathcal{T}}$	$SectionTenv \mathrel{+\mkern-14mu\rightarrow} SectionTenv$

The definition of the type rule for sections is the following:

$$<\{ section\ X\ parents\ P_1, ..., P_m \bullet Par_1, ..., Par_n \}>^{\mathcal{T}} =$$

$$\begin{aligned}
&\{ \eta: SectionTenv \mid \\
&\qquad X \notin dom\ \eta \land \\
&\qquad (\forall i : 1..m \bullet P_i \in dom\ \eta) \land \\
&\qquad (\forall\ i, j : 1..m \bullet \pi_2(\eta(P_i)) \cup \pi_2(\eta(P_j)) \in (_ \mathrel{+\mkern-14mu\rightarrow} _)) \land \\
&\qquad (\forall\ i, j : 1..m \bullet \forall\ b: \pi_2(\eta(P_i)) \cap \pi_2(\eta(P_j)) \bullet \exists\ P: \pi_1(\eta(P_i)) \cap \pi_1(\eta(P_j)) \bullet b \in \pi_2(\eta(P))) \\
&\bullet \qquad \eta \mapsto (\eta \cup \{X \mapsto (X \cup \textstyle\bigcup_k \pi_1(\eta(P_k)), \bigcup_k \pi_2(\eta(P_k)) \mathbin{;} <[Par_1]>^{\mathcal{T}} \mathbin{;} ... \mathbin{;} <[Par_n]>^{\mathcal{T}})\}) \}
\end{aligned}$$

What this says is that the section is well-typed in precisely those section type environments, η, in which the following hold:

1. The name of the new section, X, is not already declared in η.

[2] I leave it to a professional bracket designer to come up with the right shape for these things.

2. Each P_i is the name of a section already declared in η.

3. The environments associated with any two parent sections P_i and P_j are compatible, i.e., their union is a partial function. (Repetition is permitted.)

4. Any object which is common to the environments of two parent sections P_i and P_j is declared in a common ancestor of the two sections[3]. (This common ancestor or, indeed, any ancestor of any parent, may, but need not, be named in the list P_1, ..., P_m.)

5. The sequence of paragraphs, Par_1, ... Par_n, is well-typed with respect to the environment obtained by forming the union of the type environments associated with the parent sections, P_1, ..., P_m.

If the section is well-typed then the section type environment obtained by declaring it in η is η enriched to map X to a pair whose first component comprises X together with all ancestors of each parent section, P_1, ..., P_m, and whose second component is the result of declaring the sequence of paragraphs, Par_1, ..., Par_n, in the type environment obtained by forming the union of the type environments associated with the parent sections, P_1, ..., P_m.

The definition of the type rule for specifications is the following

$<(\ Sec_1,\ ...,\ Sec_n\ )>^{\mathcal{T}} = <\{Sec_1\}>^{\mathcal{T}} \fatsemi \ldots \fatsemi <\{Sec_n\}>^{\mathcal{T}}$

I.e., sections are composed just by composing their effects on the section type environment.

2.2.3 Value Rules

The value functions for sections and specifications are as shown in the following table:

Name	Form	Sort
Section Type	$<\{\ Sec\ \}>^{\mathcal{M}}$	$SectionEnv \nrightarrow SectionEnv$
Specification Type	$<(\ Spec\ )>^{\mathcal{M}}$	$SectionEnv \nrightarrow SectionEnv$

The definition of the value rule for sections is the following:

$<\{\ section\ X\ parents\ P_1,\ ...,\ P_m \bullet Par_1,\ ...,\ Par_n\ \}>^{\mathcal{M}} =$

$\{\ \eta\text{: } SectionEnv \bullet \eta \mapsto (\eta \cup X \mapsto (\bigcup_k \eta(P_k) \fatsemi <[\ Par_1\ ]>^{\mathcal{M}} \fatsemi \ldots \fatsemi <[\ Par_n\ ]>^{\mathcal{M}}))\ \}$

Again, I leave it to the relational calculus experts to recast the above in relational calculus terms. The rule is only intended to apply to sections which are well-typed and says that the section value environment obtained by declaring a section in η is η enriched to map X to the result of declaring the sequence of paragraphs, Par_1, ..., Par_n, in the value environment obtained by forming the union of the value environments associated with the parent sections, P_1, ..., P_m.

The definition of the value rule for specifications is the following

[3] See section 3.2 for the reasons for having this rule. A section is treated part of its own ancestry to simplify the statement of this rule, which would otherwise need a more complicated condition on i and j.

$$<(Sec_1, ..., Sec_n)>^{\mathcal{M}} = <\{Sec_1\}>^{\mathcal{M}} \mathbin{;} ... \mathbin{;} <\{Sec_n\}>^{\mathcal{M}}$$

I.e., as with the type rule, sections are composed just by composing their effects on the section value environment.

3 DISCUSSION

3.1 Motivation

Modularity is a pragmatic issue. I have therefore been willingly influenced by how modularity is handled by Z tools with which I am familiar. In rapidly decreasing order of familiarity, these tools are ProofPower, Balzac and Cadiz.

I have tried to identify a coherent common ground between the features offered by these tools (in so far as I understand them). From this common ground, I would claim that we get a scheme which is both efficiently implementable and simple for users to understand[4].

An objective has also been to produce a scheme which can be added as an extra layer on top of the existing semantics for paragraphs with minimum impact. There are two main reasons for this objective: *(a)*, to ease the task of adding a modularity feature to the standard; and, *(b)*, to give a rigorous framework in the style of the standard as a basis for discussing more advanced features.

3.2 Type Rules

The restriction in the type rules on objects common to the environments of two parent sections is introduced to make the modularity scheme admit efficient implementation (which, in this case, makes it easier for people to check their work too). What the rule does, for example, is prohibit the inclusion in the ancestry of a section, two sections such as:

section A parents Toolkit •

z

$x : \mathbb{N}$

$x > 957$

section B parents Toolkit •

z

$x : \mathbb{Z}$

$x > 943 \land x < 1001$

Here, the sections A and B are not allowed to coexist as ancestors of a third section even though they ascribe the same type to x. The reason is to simplify, both for people and for computers, the

[4] I do not intend to imply that any of the tools I have listed are inefficient or hard to understand. However, off the common ground, they differ, and a more complex scheme which embraced all the differences, might I think be hard to understand.

task of deciding whether or not the sections are compatible. With the proposed rules, we can see immediately that A and B are not allowed as common ancestors of a third section, because they both declare an object called x, and we do not have to worry about the type ascribed to x.

The rule does not restrict our ability to further constrain the value of an object in a descendant of the section in which it is declared.

3.3 Terminology

Existing tools and [2] use varied terminology for their modularity features. For reference, the comparison is as follows[5]:

Balzac	Cadiz	ProofPower	[2]	[1]	Proposed
Chapter	Document	Theory	Document	Library	Section
Reference	Import	Parent	Import	Inclusion	Parent

The term 'section' has been chosen in order to avoid confusion with terminology from various programming languages. The term 'parent' is pleasantly suggestive and allows useful terms like 'ancestor', 'child', and 'descendant' to be used in a fairly natural way (but, beware that the technical details of the proposal above make the ancestor relation reflexive not just transitive) . In the representation syntax, I did consider using 'uses' or 'extends' instead of 'parents', but felt that the genealogical terminology was so useful that we may as well embody it in the syntax.

3.4 Enhancements

Many people will object that the modularity scheme is very simple-minded, e.g., by comparison with that of Ada or Standard ML. I would counter that it is unfair to compare a modularity feature in a logical language with one in a programming language. Probably the enhancements for which there will be most demand are:

Local definitions: if there is a strong demand for these, then I don't think the logical problems are too great; however, for proof, what you have to do is arrange things so that the definitions of local objects are still accessible. A simple solution is to put local objects in a distinguished syntactic category (say names containing an at-sign '@') and then allow an object, '@x' say, local to a section 'm' to be accessed externally as 'm@x'. With this approach, local definitions have a modest impact on the semantics.

Parameterised sections: these require significantly more work, I believe. To be really useful, parameterised sections might have to involve a significant change to the type system, since given sets defined in a section with parameters ought to act like new type constructors with the parameters as operands (otherwise you couldn't instantiate a section with parameters more than once in a given context). Since Z already has generics, my feeling is that the need for parameterised sections is outweighed by the complexity they might introduce[6].

Operator overloading: As may be seen from the examples in the next section, things might work more smoothly for some applications, if names in one section were allowed to overload names in another section without making the sections incompatible. This can be accomodated along the

[5] Parts of this table are based on memory and hearsay, I apologise for any inaccuracies.

[6] This may be too pessimistic. The scheme proposed in [1] avoids this type of problem by requiring the user to tag each instantiation of a section with a qualifier to distinguish different instances. However, my feeling is that, if one is to have generic sections, one must be able to reason about them generically, and I am uncertain about the difficulty of supporting this with such a scheme. I would be happy to be proved wrong.

lines I've suggested for local definitions above (so that every object has a unique name including the section name, but the unadorned name may be used where appropriate). Again this adds some complexity for the standard, for the tool-builder and for a user trying to understand a specification.

Object-Orientation: I feel that modularity, as I use the term in this document, serves different needs from object-orientation. In my view, modularity deals with very pragmatic organisational concerns (e.g., reuse of libraries, separate type-checking, multiple specifiers working on one specification, etc.). Such concerns can and should be addressed independently of the conceptual style adopted for the entities within a specification (e.g., object-oriented, functional etc.). To quote [1], "[modularity and object-orientation] have different aims and should not really be seen as in competition".

4 TOOLKIT FORMAT

Using the syntax of the previous section, the toolkit as in the books [3, 4] can be presented in the following format:

section Sets
    ... definitions for sets ...
section Relations
parents Sets •
    ... definitions for relations ...
section Functions
parents Relations •
    ... definitions for functions ...
section Numbers
parents Functions •
    ... definitions for numbers ...
section Sequences
parents Numbers •
    ... definitions for sequences ...
section Bags
parents Sequences •
    ... definitions for bags ...
section Toolkit
parents Bags •
    ... no definitions needed here ...

Here, the final section, 'Toolkit', just serves to bring everything together and no definitions are needed in it.

As a more complex example, adding the real numbers can be done in two ways:

1. Assume the integers are defined in the section 'Numbers' as a subset of an unspecified given set of numbers. In this case, a section 'Reals' defining the real numbers as a superset of the integers could be added as a child of the section 'Numbers' (or any of its descendants). In this case, either the arithmetic operators on integers would have to be loosely defined so that the theory of reals can sharpen up the definitions of the existing operators, or the real operators would have to be denoted by symbols distinct from the symbols for the integer operators.

2. If the reals are to replace the integers for people who elect to use the reals, then a section 'Reals' can be added as a child of the section 'Functions'. The developer of such a section might then feel obliged to introduce custom versions of 'Sequences' and 'Bags' as descendants of 'Reals'. Such an approach might lead to somewhat smoother working with the reals themselves, at the price of not being able to reuse the material based on the integers directly.

REFERENCES

[1] Ian Hayes and Luke Wildman. Towards Libraries for Z. In *Z User Workshop, London 1992*. Springer-Verlag, 1993.

[2] S. King, I.H. Sorensen, and J. Woodcock. Z: Grammar and Concrete and Abstract Syntaxes. *Programming Research Group, University of Oxford*, 1987.

[3] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[4] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice-Hall, 1992.