Towards Libraries for Z

Ian Hayes and Luke Wildman
Department of Computer Science,
University of Queensland

February 9, 2023

Abstract

We consider adding parametrised libraries to Z as a strict extension to the current notation. We examine a simple modularisation facility with only generic sets as parameters, similar to current Z generic schemas.

In examining parameters other than generic sets we consider both an explicit parameter section at the beginning of the library and a more general alternative allowing any variable in the library to be instantiated as a parameter. It turns out, however, that the same effect as the latter form of parametrisation can be achieved with just the simple modularisation, that is, with only generic set parameters.

Finally, we consider interactions between the modularisation facility and current Z notation. In particular, we consider the problem of allowing flexibility in using free types and schema types as parameters.

1 Introduction

The objective of this paper is to examine the possibilities and problems involved with adding parametrised libraries to Z [Spi92]. Our aim is to preserve as much as possible the basic Z language, so that modularisation becomes a straightforward language extension.

We would like the modularisation facility to be usable with all forms of Z specifications, not just those making use of the conventions for specifying sequential programs, (that is, specifications written in terms of a state and a number of state-to-state operations using conventions such as primed and unprimed variables, delta schemas, etc.). Hence our approach differs from that of Object-Z [DKRS91], OOZE [AG91], Z^{++} [Lan91], and MooZ [MC91] (see [ZIP91] for a comparison of these). These introduce modularisation features centred around modelling abstract data types with state and operations. For this paper, we see a modularisation facility as a way of grouping together, and perhaps parametrising, a number of definitions, in the same way that a schema groups together components. A library can thus be viewed as a super-schema. The two approaches have different aims and should not really be seen as in competition. Our work is closer to that of Duke [Duk91b, Duk91a].

Currently, Z has little in the way of facilities for modularisation.

- One can refer to definitions in a library.
- One can define a schema to collect together a number of components, along with a predicate linking these components. These components can only be variables, not nested definitions.

In both cases the definitions may be generic and the formal parameters are unstructured sets. The actual parameters may be structured sets. However, no knowledge of this structure is available within the library.

In Section 2 we present a simple modularisation facility, which can be viewed as an extension of generic schemas. Only (unstructured) sets are allowed as parameters to these libraries. These simple facilities are quite useful in practice and allow, for example, a well-structured presentation of the CAVIAR specification as given in [FH87]. In Section 3 we consider parameters other than unstructured sets, and dependencies between parameters. These lead to some interactions with other Z notation. These problems are discussed in Section 4.

2 Simple modularisation

2.1 No parameters

The simplest form of modularisation is just to allow the collecting together of a set of definitions into a single library. Mathematical libraries, such as sequences, bags, etc., are commonly used with Z specifications, so it is essential that a modularisation facility at least support this requirement. This is not difficult.

As a Z specification is viewed as a document it is appropriate to view a library as a chapter or section of a specification document. In fact, the term chapter has been used for such libraries in Z for many years.

One problem with using such libraries is that, especially in the case of libraries developed independently, the same name may be used in more than one library. If one would like to use these libraries together within a specification, then there needs to be some way of disambiguating references to a common name. A simple way of doing this is to qualify references to common names with the library's name to make the reference unambiguous (assuming of course that the libraries themselves have been given different names).

This ability to create separate name spaces and disambiguate references to names is the first advantage of having a modularisation facility. With the possibility of multiple instantiations of the same library, disambiguation of names becomes more important.

2.2 Z-style generic parameters

The next level of facility we consider is allowing generic libraries. The parameters allowed are unstructured sets, similar to those currently allowed in Z for generic definitions. This allows us to group together a collection of related definitions *generic as a whole* in the parameter sets. When such a library is used, its generic sets need

to be instantiated. A single instantiation of the library provides all the definitions within the library instantiated with the actual parameter sets.

2.2.1 Z generic definitions

As with Z generic definitions, different instantiations need to be distinguished. With current Z generics, the context of an instantiation often uniquely determines the parameter set, but in general the parameter set may need to be specified explicitly. For example, given a generic function *length* defined by

$$[X]$$
$$length : \operatorname{seq} X \longrightarrow \mathbb{N}$$
$$length(s) = \#s$$

then for $s : \operatorname{seq}\mathbb{N}$ in the context '$length(s)$', *length* is taken to mean $length[\mathbb{N}]$ as its type is uniquely determined by the type of s. However, in the context '$length(\langle\rangle)$' the type of the empty sequence, $\langle\rangle$, cannot be uniquely determined, and hence the type of *length* cannot be uniquely determined. In this case we can write $length[\mathbb{N}](\langle\rangle)$ to fully determine the type of *length*.

For the current generic definitions in Z, there is a requirement that the object being defined is uniquely determined by the definition for any given actual parameter set(s). This requirement was introduced because every use of a generic definition is considered to be a separate instantiation of that definition (see [Spi92] for more details). As the objects being defined may be used within expressions, two instances of a definition with the same parameter set(s) are required to have the same value in order to maintain referential transparency. In the use of such a generic definition, as the actual parameters uniquely determine its value, no further disambiguation is required.

2.2.2 Generic libraries

While the approach of requiring each generic definition to be uniquely defined for every actual generic parameter is appropriate for single generic definitions, for the case of a generic library containing a number of *variable declarations* as well as definitions, the same approach is not suitable.

First and foremost one would like to be able to include variables within a generic library whose values are not uniquely determined by the generic set parameters. Different instantiations of the same library, even with the same generic parameters, could then use different values for their respective variables. A not uncommon mistake in Z specifications is to write generic definitions of variables that are not uniquely determined by the generic sets. We would like to provide a facility for doing this correctly.

Secondly, if one separates the instantiation of a library from the use of that instantiation's definitions, then there is only one instantiation and the reason for the uniqueness condition for Z generics disappears.

For multiple instantiations of the same library one needs to be able to distinguish variables in one instantiation from variables in another. This can be achieved straightforwardly by providing a mechanism for naming instantiations. We shall do this by prefixing the instantiation with a name followed by '::'. This not only allows us to distinguish different instantiations of a definition without having to supply the generic sets, but also allows us to distinguish different instantiations with the same generic sets.

2.2.3 Example: grammar library

As an example let us consider a library useful for dealing with context-free grammars. The library is generic over the symbol set and has four variables defining the grammar: the terminal symbols, the nonterminal symbols, the set of productions and the starter symbol. Within the library from which this example was taken, a number of useful derived variables determined from the grammar are specified. Here we limit ourselves to the single derived variable *derives*.

The library appears as Section 2.2.4 below. The title of the section consists of '**Library:**' followed by the name of the library, followed by an optional list of generic set formal parameters in square brackets.

Within a library, a formal parameter set acts like a basic type in a normal Z specification. The scope of variables and schemas defined within a library is limited to the library. However, a specification (or another library) may gain access to the definitions within a library by instantiating the library.

When a library is instantiated, actual parameter sets are specified. For example, the *GrammarOps* library of Section 2.2.4 can be instantiated with the actual set of symbols AS by the declaration,

$$GrammarOps[AS].$$

All the definitions of the library *GrammarOps* are available within the specification that included the instantiation. However, every occurrence of a formal parameter within the library definition is replaced by the corresponding actual parameter set within the instantiation.

When a library is instantiated, its name may also be qualified to distinguish definitions with the same name in different libraries. The qualification consists of a name followed by '::'. For example, we may instantiate *GrammarOps* with qualifier A as follows.

$$A::GrammarOps[AS]$$

The qualifier is applied to all names defined in the library. We discuss the special case of qualification of schemas in Section 3.2.

2.2.4 Library: GrammarOps[S]

This library is parametrised by the set of symbols to be used with the grammar, S. The types of the four variables that define the grammar are all defined in terms of this generic symbol set, S.

$$
\begin{array}{|l}
productions : S \leftrightarrow \operatorname{seq} S \\
terminals, nonterminals : \mathbb{P}\, S \\
starter : S \\
\hline
starter \in nonterminals \\
terminals \cap nonterminals = \{\} \\
\operatorname{dom} productions \subseteq nonterminals \\
(\forall\, s : \operatorname{ran} productions \bullet \operatorname{ran} s \subseteq terminals \cup nonterminals)
\end{array}
$$

In order for the grammar to be valid these variables satisfy a set of constraints: the starter symbol must be a nonterminal, the terminal and nonterminal symbols should be disjoint, only nonterminals may appear on the left side of a production, and only terminals and nonterminals may appear on the right side of a production.

The *derives* relation is derived from the grammar as follows: if a nonterminal N can produce the string of symbols (terminals and nonterminals) β, then a string of symbols containing N can derive the same string with N replaced by β.

$$
\begin{array}{|l}
derives : \operatorname{seq} S \leftrightarrow \operatorname{seq} S \\
\hline
\forall\, \alpha, \beta, \gamma : \operatorname{seq} S;\ N : nonterminals \bullet \\
\qquad (\alpha \frown \langle N \rangle \frown \gamma, \alpha \frown \beta \frown \gamma) \in derives \Leftrightarrow (N, \beta) \in productions
\end{array}
$$

3 Parameters other than generic sets

The parametrisation mechanism discussed above only allows unstructured sets as parameters. The next logical step is to allow parameters other than generic sets. We can allow values including (structured) sets as parameters and allow dependencies between parameters to be specified. We considered two approaches to providing more general parameters:

- having an explicit parameters section at the head of a library to specify the types of and constraints on the parameters, and
- allowing any of the variables declared in the library to be a parameter.

The first approach allows the simple syntax of positional parameters, but fixes once and for all the parameters to the library. If partial instantiation of parameters is required — because not all the parameters are known at the point of initial instantiation — a special mechanism needs to be introduced.

In the second approach any variable in the library may be regarded as a parameter and instantiated to a value. This requires the use of a keyword parameter mechanism on instantiation, but in allowing any of the variables in a library to be considered as parameters it treats partial instantiation as the norm.

The second approach is appealing as it would appear to allow libraries to be used in contexts not previously considered by the author of the library, using a different set of parameters to those used originally, without the necessity to revise the definition of the library. This leads to the library being more general than a statically parametrised library and able to be *re-used* in more contexts.

From the preceding discussion it is clear that we prefer the second, more flexible, approach outlined above. The real surprise is that one can achieve the same overall effect as the second approach with the simple generic libraries of Section 2! One instantiates a library with appropriate sets for the generic set parameters. Any of the variables of the library can then be effectively instantiated to a value by simply adding a normal Z predicate equating the variable to the value. In fact, this approach is more general: the predicate need not be an equality, but may just further constrain the value of the variable. So, we get not only partial instantiation in the sense of only *equating* some of the variables to be particular values, but also in the sense of only *constraining* some of the variables. The generality and simplicity of this approach are appealing.

For this approach the grammar library is as in Section 2.2.4, that is, with only generic set parameters. The library can be instantiated with its generic set:

$$A :: GrammarOps[AS],$$

and then any parameters are effectly instantiated by adding normal Z predicates after the library instantiation. For example, the variables describing a grammar may be instantiated as follows.

$$A :: productions = Aprod$$
$$A :: terminals = Aterm$$
$$A :: nonterminals = Anonterm$$
$$A :: starter = Astart$$

A 'with' construct This last predicate giving the values for the grammar variables requires multiple uses of the qualifier '$A::$'. This repetition can be avoided by the introduction of a *with* construct similar to that available in many programming languages. The above predicate can then be written

with $A \bullet$
$$productions = Aprod$$
$$terminals = Aterm$$
$$nonterminals = Anonterm$$
$$starter = Astart$$

Renaming on instantiation Another useful facility is the ability to change the name of variables or definitions within a module on instantiation of the module. This can be provided by extending the renaming capability for Z schemas to apply to libraries as well. This is straightforward.

3.1 An example: A resource-user system

As a second example we consider a resource-user system parametrised over three generic sets: *T, R, U*. This example is a simplified version of the resource-user system in the CAVIAR specification [FH87]. Informally, T is to be thought of as a set of *time slots*, R is a set of *resources* and U is a set of *users*. We describe a

general resource-user system as a function from T to the set of relations between R and U. Thus we have a rather general framework: for each time slot $t \in T$, some users are occupying or using some resources. The specification of the resource-user system follows as Section 3.1.1.

3.1.1 Library: ResourceUser[T,R,U]

The state of a resource-user system is modelled by a function ru which for every time t gives the relationship between resources and users at that time.

State

$$ru : T \longrightarrow (R \leftrightarrow U)$$

This state allows both multiple users of a resource, and a user to have multiple resources.

The *initial state* of this system is defined by making $ru(t)$ the empty relation for each t.

$$Init \mathrel{\widehat=} [State \mid \operatorname{ran}(ru) = \{\{\}\}]$$

A resource $r?$ may be booked by user $u?$ for all the times in the set $t?$ provided the user has not already booked that resource for any of those times.

Book

$$\begin{array}{l} \Delta State \\ t? : \mathbb{P}\ T \\ r? : R \\ u? : U \\ \hline \forall\, t : t? \bullet \\ \quad (r? \mapsto u?) \notin ru(t) \land \\ \quad ru'(t) = ru(t) \cup \{r? \mapsto u?\} \\ t? \lhd ru' = t? \lhd ru \end{array}$$

The resource-user relation is updated to record the booking for only those times in $t?$.

A user $u?$ may cancel a booking for a resource $r?$ for a set of times $t?$, provided the user has the resource booked for all those times.

Cancel

$$\begin{array}{l} \Delta State \\ t? : \mathbb{P}\ T \\ r? : R \\ u? : U \\ \hline \forall\, t : t? \bullet \\ \quad (r? \mapsto u?) \in ru(t) \land \\ \quad ru'(t) = ru(t) \setminus \{r? \mapsto u?\} \\ t? \lhd ru' = t? \lhd ru \end{array}$$

The resource-user relation is updated to remove the bookings for only those times in $t?$.

3.2 Decoration of components of schemas

The resource-user library can be instantiated for different types of resources and users and even different types of times. In the CAVIAR specification there are multiple instantiations of this library in different guises. For illustration we consider two instantiations:

- reservations of hotel rooms, HR, for visitors, V, on dates, $Date$, and
- reservations for transport, TR, for visitors, V, at times, $Time$.

The instantiations follow.

$$HR\_V :: ResourceUser[Date, HR, V]$$
$$TR\_V :: ResourceUser[Time, TR, V]$$

(Please ignore the fact that multiple bookings of a room for the same date are allowed; this is avoided in the full CAVIAR specification, but ignored here for simplicity.) The above instantiations give us two state schemas, $HR\_V :: State$ and $TR\_V :: State$. For the full CAVIAR system we need to combine both these states (and more) together into a system state schema. We can write,

CAVIAR_State

$$HR\_V :: State$$
$$TR\_V :: State$$

However, we need to be careful how we interpret a reference like $HR\_V :: State$:

- We can interpret it to mean that just the name of the schema is qualified. In which case it is equivalent to

  HR_V::State

  $$ru : Date \longrightarrow (HR \leftrightarrow V)$$

- Alternatively, the qualification '$HR\_V ::$' can be used as a decoration on the schema components as well. In which case it is equivalent to

  HR_V::State

  $$HR\_V :: ru : Date \longrightarrow (HR \leftrightarrow V)$$

If we take the former meaning then the definition of *CAVIAR_State* is invalid as it attempts to merge two incompatible ru components. For the second interpretation, the names $HR\_V :: ru$ and $TR\_V :: ru$ are distinct and the definition of *CAVIAR_State* is valid. Further, as the schema component names are distinct, operations defined by

combining hotel reservation and transport reservation operations are well defined. For example, an operation to book a hotel room and leave the transport reservations unchanged can be defined by

$$Book\_HR \mathrel{\widehat=} HR\_V::Book \land TR\_V::\Xi State.$$

If we reconsider the first alternative (qualification but no decoration), then we need to distinguish the hotel and transport reservations. This could be done by declaring two variables of the appropriate type of state:

CAVIAR_State

$$hr : HR\_V::State$$
$$tr : TR\_V::State$$

The problem now becomes defining operations on this state, such as an operation to book a hotel room. In order to do this we need to define an operation which performs a booking operation on the *hr* component of the above state:

Book_HR

$$\Delta CAVIAR\_State$$
$$t? : \mathbb{P}\ Date$$
$$r? : HR$$
$$u? : V$$

$$\exists \Delta HR\_V::State \bullet$$
$$\quad \theta HR\_V::State = hr \land$$
$$\quad HR\_V::Book \land$$
$$\quad hr' = \theta HR\_V::State'$$

This definition is more complicated than the version given above which assumes decoration of schemas.

While automatically decorating the components of a schema with a library qualifier works well for the example given above, it is not necessarily the preferred approach in general. It requires that component names are always decorated even when there are no clashes and decoration is not necessary, and it also precludes the merging of components of schemas from two separate libraries if this were desired.

One approach to having a relatively simple specification of such operations without automatically decorating the components of a schema with the library qualifier, is to allow explicit decoration of schemas. The CAVIAR state could then be written

CAVIAR_State

$$hr\text{-}HR\_V::State$$
$$tr\text{-}TR\_V::State$$

where '*hr-*' is a decoration, but '*HR_V*' is only a name qualifier. This expands to

CAVIAR_State

$$hr\text{-}ru : Date \longrightarrow (HR \leftrightarrow V)$$
$$tr\text{-}ru : Time \longrightarrow (TR \leftrightarrow V)$$

The booking operation can then be written as follows.

$$Book\_HR \mathrel{\widehat=} hr\text{-}HR\_V :: Book \land tr\text{-}TR\_V :: \Xi State.$$

This approach has the appeal that it separates the concerns of qualification and decoration. These two notational devices become primitives that may be used either independently or combined together to create the effect desired by the specifier. Although this approach is slightly more cumbersome than automatic decoration of schema components, it is no where near as complicated as having no decoration at all.

4 Structured set parameters

One area of interest is if a library requires some knowledge of the internal structure of a generic set parameter. For example, the library may assume that the generic set is a cartesian product of two sets X and Y. This case can be handled simply by making both X and Y generic set parameters, rather than their cartesian product. The cartesian product can then be defined in terms of X and Y within the library. Of more interest are the cases when the generic set is expected to be a free type (Section 4.1) or a schema type (Section 4.2).

4.1 Free type

A free type may be used as part of a library. For example, the free type may be of the form,

$$T ::= x\langle\langle X \rangle\rangle \mid y\langle\langle Y \rangle\rangle,$$

where the sets X and Y may or may not be generic set parameters to the library.

In some cases, however, one would like the free type T to be treated like a parameter and to be equated with a subset of some existing free type which has constructors of the same type as x and y, but may also have other constructors. For example, consider the set $T2$

$$T2 ::= x2\langle\langle X \rangle\rangle \mid y2\langle\langle Y \rangle\rangle \mid z2\langle\langle Z \rangle\rangle,$$

which is similar to T. It can be thought of as a superset of T.

To see how we can achieve this it is instructive to expand the free type definition of T above. T is introduced as a basic type:

$$[T],$$

with two total one-to-one constructor functions x and y. The ranges of these constructor functions are disjoint (1a) and together they construct the whole of T (2a).

$$\begin{array}{|l} x : X \rightarrowtail T \\ y : Y \rightarrowtail T \\ \hline \mathsf{disjoint}\ \langle \operatorname{ran} x, \operatorname{ran} y \rangle \qquad (1a) \\ \bigcup \{\operatorname{ran} x, \operatorname{ran} y\} = T. \qquad (2a) \end{array}$$

In this form it is clear that if we equate T with $T2$, x with $x2$ and y with $y2$, that constraint (2a) is not satisfied. However, if we remove constraint (2a), we can make T a generic parameter which we can instantiate with $T2$, and then equate x with $x2$ and y with $y2$ to achieve the desired effect. So, by using the expanded form of the definition of T without constraint (2a), we can achieve our objective.

For free types within a library that we wish to allow to be substituted by 'super' types, we could allow the following syntactic sugar as a predicate within a library:

$$T \supseteq x\langle\langle X \rangle\rangle \mid y\langle\langle Y \rangle\rangle.$$

T would be a generic parameter, and x and y can be equated with the actual constructor functions.

Recursive free types The principle used above may be applied to free types whose definitions involve recursion, however, the expansion of the free type definition in this case involves a more general version of condition (2a). We consider a simplified example of abstract syntax for boolean expressions and assertions taken from a specification of a simple programming language.

$$BE ::= const\langle\langle Boolean \rangle\rangle \mid var\langle\langle Id \rangle\rangle \mid and\langle\langle BE \times BE \rangle\rangle$$

The expansion of this definition introduces the basic type BE,

$$[BE]$$

and defines the constructor functions *const*, *var* and *and*:

$$\begin{array}{l} const : Boolean \rightarrowtail BE \\ var : Id \rightarrowtail BE \\ and : BE \times BE \rightarrowtail BE \\ \hline \mathsf{disjoint}\ \langle \mathrm{ran}\ const, \mathrm{ran}\ var, \mathrm{ran}\ and \rangle \qquad (1b) \\ \forall\, W : \mathbb{P}\, BE \bullet \qquad (2b) \\ \quad const(\!|Boolean|\!) \cup var(\!|Id|\!) \cup and(\!|W \times W|\!) \subseteq W \\ \quad \Rightarrow BE \subseteq W. \end{array}$$

A 'superset' of boolean expressions are assertions with the following abstract syntax:

$$\begin{aligned} Assertion ::= {} & Aconst\langle\langle Boolean \rangle\rangle \mid Avar\langle\langle Id \rangle\rangle \mid Aand\langle\langle Assertion \times Assertion \rangle\rangle \\ & \mid forall\langle\langle Id \times Assertion \rangle\rangle. \end{aligned}$$

To pass *Assertion* for BE we need to drop condition (2b) and equate the corresponding constructor functions.

Note that the constructors *Aconst*, *Avar* and *Aand* do not generate all possible assertions. However, BE has been instantiated to the set of all assertions, so *Aand* is of type

$$Aand : Assertion \times Assertion \rightarrowtail Assertion.$$

The arguments to this constructor may be any assertions, including assertions not generated solely with the three constructor functions, ie, they involve the *forall* constructor. Care should be taken within the library as to whether such constructed objects are desirable or not.

4.2 Schema type

As with free types it is useful if a library may expect the structure of a set passed to it to be a schema with certain components. For example, a library may expect a parameter to be of the form

$$
\begin{array}{|l}
S \\
\hline
c : C \\
d : D \\
\hline
P(c, d) \\
\hline
\end{array}
$$

where P is some predicate possibly involving c and d, and the sets C and D may or not be generic set parameters. It is useful to allow sets that are *similar* to S to be passed as parameters as well. For example, consider the schema $S2$ similar to S.

$$
\begin{array}{|l}
S2 \\
\hline
c2 : C \\
d2 : D \\
e2 : E \\
\hline
P(c2, d2) \land Q(c2, d2, e2) \\
\hline
\end{array}
$$

As with free types, it is useful to break the schema S down into an *almost* equivalent form. S is considered to be a basic type with two total selector functions c and d and an onto constructor function mkS.

$$[S]$$

The components $c(s)$ and $d(s)$ of any element of $s \in S$ satisfy the predicate P (3), and for any pair, $cx : C$ and $dx : D$ that satisfy P, a corresponding element of S exists (4) and this element is unique (5). The constructor function mkS can build any element of S (it is onto) given a pair cx and dx satisfying P (6), and the c and d components of such a constructed element are cx and dx respectively (7). (There is quite a deal of redundancy in the conditions (4)–(7), but this is useful to keep the concepts involved separate.)

$$
\begin{array}{|ll}
c : S \longrightarrow C & \\
d : S \longrightarrow D & \\
mkS : C \times D \pfun S & \\
\hline
(\forall\, s : S \bullet P(c(s), d(s))) & (3) \\
(\forall\, cx : C; dx : D \mid P(cx, dx) \bullet & \\
\quad (\exists\, s : S \bullet cx = c(s) \land dx = d(s))) & (4) \\
(\forall\, s1, s2 : S \bullet & \\
\quad c(s1) = c(s2) \land d(s1) = d(s2) \Rightarrow s1 = s2) & (5) \\
\dom mkS = \{cx : C; dx : D \mid P(cx, dx)\} & (6) \\
(\forall\, cx : C; dx : D \mid P(cx, dx) \bullet & \\
\quad c(mkS(cx, dx)) = cx \land d(mkS(cx, dx)) = dx) & (7)
\end{array}
$$

If we are to allow schemas with additional fields, such as $S2$ to be substituted in place of S, then we can no longer insist on the uniqueness condition (5) — because two elements of $S2$ with the same $c2$ and $d2$ fields may have different $e2$ fields — nor can we make use of the constructor function mkS (plus (6) and (7)) — because we need to know the value of component $e2$ to construct a value of $S2$. Provided $S2$ satisfies conditions (3) and (4), we can instantiate S to $S2$ and the selector function c to $(\lambda\, S2 \bullet c2)$ and d to $(\lambda\, S2 \bullet d2)$.

If $S2$ puts a tighter constraint on fields $c2$ and $d2$ (via predicate Q) than S does, then we cannot insist that that there exist a value of s corresponding to every pair $cx : C$ and $dx : D$ such that $P(cx, dx)$ holds, thus eliminating condition (4) as well. In this case the declarations within the library become

$$\begin{array}{|l} c : S \longrightarrow C \\ d : S \longrightarrow D \\ \hline \forall\, s : S \bullet P(c(s), d(s)) \end{array}$$

and we can substitute $S2$ for S as above. As these declarations are quite straightforward, we will not bother introducing any syntactic sugar for this case.

Although the above allows compatible schemas to be treated as parameters, it only allows the schemas to be treated as sets. It does not allow for schema operations. It is not clear how one would go about allowing schema operations in such a context.

5 Conclusions

By far the most significant conclusion of this work is that parametrised libraries may be added to Z by simply adding libraries with generic set parameters. The effect of parameters other than generic sets may be achieved by equating variables declared within an instantiation of a library with the desired value. In fact, the mechanism is more general than explicit parametrisation, as we may just place constraints on the value of the variable, rather than giving an explicit value. In addition, any of the variables declared within a module can be thought of as a parameter. This provides a more flexible modularisation and parametrisation mechanism than statically predetermined parameters. We envisage that this could lead to easier reuse of libraries.

It was a relevation to us that such a simple mechanism could provide such a flexible modularisation facility.

The approach that we have taken with schemas has been to only qualify the name of the schema with the library's qualifier and not decorate the schema's components with the qualifier. The problem of combining schemas to represent an operation on states made up from multiple instantiations of a library is solved by allowing schemas to be decorated with a name as described in Section 3.2. This approach has been taken because it separates the concerns of qualification of names and decoration of schemas. Providing these two facilities separately allows the specifier the flexibility of combining them or using them independently as needed.

If a library needs some knowledge of the internal structure of a generic parameter set, this can be provided. The more interesting cases are free types (Section 4.1)

and schema types (Section 4.2). Access to the structure of these types can be provided via constructor functions and selector functions, respectively. In general, one would like to be able to supply compatible sets as parameters by relaxing some of the conditions of the normal definitions. This can already be done by simply writing out the required conditions explicitly, but may be aided by new notation to help express the desired constraints more succinctly. This is an area for further investigation.

The libraries presented here can be thought of as super schemas. Or, put the other way around, a schema can be thought of as a mini library. In fact, there are no difficulties in allowing a schema to be instantiated just like a library. We would recommend allowing such instantiations. The main differences between schemas and libraries are that schemas can be used as types and can be combined using schema operators. Neither of these two facilities is applicable to libraries.

One point that has not been discussed so far is the inclusion of the same library in unqualified form more than once in the same specification. For example, a mathematical library (like bags) could be included in two separate libraries. If both of these libraries are included (unqualified) then the mathematical library definitions have been included more than once in the same specification. As both these inclusions derive from the same source it seems reasonable to merge their definitions, rather than treat this as an error. The same approach would not be recommended for two definitions with the same name that derive from different sources. For further discussion on this point see [Fit91].

6 Acknowledgements

This work has benefited from discussions with Mike Spivey, Bill Flinn and Ib Holm Sørensen on the modularisation of the CAVIAR specification, and from the work of David Duke [Duk91b, Duk91a]. Luke Wildman is supported by an Australian Postgraduate Research Award. We would also like to acknowledge our collaboration with Jim Welsh and David Carrington on an Australian Research Council supported grant entitled *Modularity in the Derivation of Verified Software* (A48931426).

References

[AG91] Antonio J. Alencar and Joseph A. Goguen. OOZE: An Object-Oriented Z Environment. In Pierre America, editor, *ECOOP'91: European Conference on Object-Oriented Programming, volume 512 of Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 1991.

[DKRS91] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. The Object-Z specification language version 1. Technical Report SVRC 91-1, Department of Computer Science, University of Queensland, QLD 4072, Australia, May 1991.

[Duk91a] D. Duke. Enhancing the structure of Z specifications. In John E. Nicholls, editor, *Proceedings of the Sixth Annual Z user Meeting*, Workshops in Computing. Springer-Verlag, 1991.

[Duk91b] D. Duke. Structuring Z specifications. In *Proceedings of the 14th Australian Computer Science Conference*, 1991.

[FH87] Bill Flinn and Ib Holm Sørensen. CAVIAR: A case study in specification. In I. J. Hayes, editor, *Specification Case Studies*, pages 141–188. Prentice Hall International, 1987.

[Fit91] John S. Fitzgerald. *Modularity in model-oriented formal specifications and its interaction with formal reasoning*. PhD thesis, Department of Computer Science, University of Manchester, 1991.

[Lan91] Kevin C. Lano. Z^{++}, an object-orientated extension to Z. In John E. Nicholls, editor, *Proceedings of the Sixth Annual Z user Meeting*, Workshops in Computing. Springer-Verlag, 1991.

[MC91] Silvio Lemos Meira and Ana Lúcia C. Cavalcanti. Modular object-oriented Z specifications. In John E. Nicholls, editor, *Proceedings of the Sixth Annual Z user Meeting*, Workshops in Computing. Springer-Verlag, 1991.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.

[ZIP91] ZIP Technical Review Committee. *Comparative Study of Object Orientation in Z*. Logica UK Ltd, November 1991.