Type-constrained Generics for Z

Samuel H. Valentine[1], Ian Toyn[1], Susan Stepney[2], and Steve King[1]

[1] Department of Computer Science, University of York, UK
{sam,ian,king}@cs.york.ac.uk
[2] Logica UK Ltd, Betjeman House, 104 Hills Road, Cambridge CB2 1LQ, UK
stepneys@logica.com

Abstract. We propose an extension to Z whereby generic parameters may have their types partially constrained. Using this mechanism it becomes possible to define in Z much of its own schema calculus and refinement rules.

1 Introduction

The Z notation [5, 2, 7] has a type system based on given sets and on generic types. Powerset and product constructors (Cartesian and schema) form new types from other types. Nothing else is a type.

The resulting system is useful, flexible and decidable. Generic types allow the definitions of many relations and functions to be made in the mathematical toolkit. The function of set union, for example, can be defined thus:

$$\_ \cup \_ [X] == \lambda S, T : \mathbb{P}\, X \bullet \{x : X \mid x \in S \lor x \in T\}$$

To use this definition, the generic parameter X can be instantiated to a set of any type whatever. Generic types are considered as atomic in themselves. No constraint is placed on the sets which instantiate generic parameters.

Whereas this works fine for set operations like union, there are other cases to which it does not extend, such as functions referring to arguments which are known to be schemas, but with unknown signatures, or known to be Cartesian products, but of unknown size. For example, the predicate

$S \lor T = \neg [\neg S; \neg T]$

is a tautology whenever it is well-typed, and one might wish to formulate it as a theorem, or even to use it to frame a definition of schema disjunction.

In Z as it stands, any such attempt would be a type error, since it would require S and T to be of compatible schema types, whereas if the types of S and T are given by generic parameters, we cannot constrain them. Yet the intention is fairly clear and the ability to define things in this way would be useful, since it would allow the existing schema calculus to be defined explicitly in Z, would provide a facility for users to extend it, and would allow the statement and use of general lemmas about schemas.

For features of the Z notation such as schema composition, existing definitions and statements of proof rules either:
a) translate into equivalent Z using a translation process stated more or less informally, such as that given in [5] or [7], or
b) omit rules about schema composition and features of similar difficulty altogether [11, 1, 4].

The purpose of this paper is to show how we can express most of schema calculus, including schema composition and piping, in Z, provided we introduce "undecoration" and allow type constraints on generic parameters.

2 Implicit Instantiation - Schema Negation

Generic definitions in Z can be written in either of two ways. One way is designed for explicit instantiation, where the instantiation can be to any set, as in an expression like $\mathbb{F}\, S$. The other is to create a generic object whose type can be inferred from the context of its use, for example where the type of a function is inferred from the type of its argument or of its result. Throughout this paper we shall work to the latter convention, where definitions are designed on the assumption that the generic parameter need not be instantiated explicitly, and is a type.

As a first example, let us examine schema negation. As we have observed in [10], we can define schema negation as set complementation, by saying

$$schNeg\ [X] == \lambda\, S : \mathbb{P}\, X \bullet \{x : X \mid \neg\, x \in S\}$$

This definition applies to any set, rather than being restricted to schemas as the standard schema negation is.

One of the innovations of the proposed Standard for Z [7] is a fixed syntax for conjectures. Using this, we can pose conjectures about negation, such as

$$[X] \models ?\ \forall\, S : \mathbb{P}\, X \bullet schNeg\ (schNeg\ S) = S$$

3 Type Compatibility - Schema Conjunction

To define schema conjunction, we could most briefly say

$$\_\ schConj\ \_\ [X, Y] == \lambda\, S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet [S;\ T]$$

It is clear that S and T must be schemas, since they have been used as declarations. In order to be valid they must also be type-compatible. That is, if we had declared $S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y$, where X and Y are types, then where X and Y have the same component names, the corresponding types must be identical. In [7] this condition would be written as $X \approx Y$. This constraint is implicit in the

definition given, so we could relax the rules and allow that form of definition as it stands.

It seems preferable, however, to allow the existing type checks to be imposed on generic parameters used under the existing rules. We therefore propose a new bit of syntax to introduce generic parameters that may be subject to type constraints. We do this by putting a † in the generic parameter list. Generic parameters preceding the † are subject to the existing rules, and may not be constrained. Generic parameters following the † may be subject to type constraints.

It seems likely that the idea would extend to any constraints, such as the case where the parameter is known to be a Cartesian product, but of unknown size. The only cases we consider in detail in this paper, however, are those where the parameters are constrained to be schemas, and those are the only cases for which we have worked out the implications for implementation in a tool [8].

So we now write

$$\_ \mathit{schConj} \_ [\dagger X, Y] == \lambda S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet [S;\ T]$$

We can pose conjectures about the functions we have defined, such as

$$[\dagger X, Y] \models? \operatorname{dom}(\_ \mathit{schConj} \_) = \mathbb{P}\, X \times \mathbb{P}\, Y$$

$$[\dagger X, Y, Z] \models? \\ \quad \forall S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y;\ U : \mathbb{P}\, Z \bullet \\ \quad\quad (S\ \mathit{schConj}\ T)\ \mathit{schConj}\ U = S\ \mathit{schConj}\ (T\ \mathit{schConj}\ U)$$

$$[\dagger X] \models? \\ \quad \forall S : \mathbb{P}\, X \bullet (S\ \mathit{schConj}\ \mathit{schNeg}\ S) = [X \mid \mathit{false}]$$

4 Syntactic Overloading - Schema Logical Operations

In a similar way, we could define schema disjunction

$$\_ \mathit{schDisj} \_ [\dagger X, Y] == \lambda S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet \mathit{schNeg}\ [\mathit{schNeg}\ S;\ \mathit{schNeg}\ T]$$

schema implication

$$\_ \mathit{schImp} \_ [\dagger X, Y] == \lambda S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet \mathit{schNeg}\ [S;\ \mathit{schNeg}\ T]$$

and schema equivalence

$$\_ \mathit{schEquiv} \_ [\dagger X, Y] == \lambda S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet [S\ \mathit{schImp}\ T;\ T\ \mathit{schImp}\ S]$$

In Z as it stands [7], the same symbols are used for predicate disjunction etc. as for schema disjunction etc. That is why, in the interests of clarity, the five definitions above have been made in terms of separate names. The precedences of schema disjunction etc. are also out of the range allocated to user-defined functions. Apart from these syntactic considerations, the above definitions could be used in the Mathematical toolkit of Z to define the operations concerned. Whereas the range of the precedences could be modified, the overloading of the names of the symbols seems to be a deeper issue. We proceed therefore on the assumption that the five operations concerned remain defined in the core language, but that the ideas and notation developed above will be used for the other operations of the schema calculus as described below.

5 Schema Axiomatic Definitions

The definitions above have all been in the explicit horizontal form. If the axiomatic form of definition is used, similar principles apply, but the question arises as to whether the type constraints on the generic parameters imposed implicitly by some of the predicates given in the axiomatic box necessarily apply to all other predicates in the same box. We answer this question in the affirmative, since if we wish the various parts to be considered separately, we can use more than one box.

6 The scope of Generic Schemas

An alternative definition of schema disjunction might be something like:

$$\_\, schDisj2 \,\_\, [\dagger X, Y] == \lambda\, S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet [X;\ Y \mid \theta X \in S \lor \theta Y \in T]$$

A difficulty with this is that any of the names X, S, Y or T might be a component name of one (or both) of the schemas which instantiate X and Y. Under the current scope rules the terms θX, S, θY and T must be interpreted in the context created by the declaration $X;\ Y$, and if any of the names X, S, Y or T was a component name of one of them, this would destroy the intended meaning of the whole definition.

We therefore stipulate that names brought into scope by being the component names of schemas that are instantiations of generic parameters, belong to a different name-space from other names. Thus the component names of the schemas that instantiate X and Y above, and the names generated by the terms θX and θY, are in a different name-space from the stated names X, S, Y or T themselves.

Explicit uses of names, such as those of X, S, Y or T above, must refer to declarations which are not dependent on the instantiations of the generic parameters of the containing paragraph. Declarations of both sorts may occur, that is, uses of schema generic parameters as declarations, and ordinary non-generic

or fully instantiated declarations. If they occur within the same schema-text, in order that the type system can remain consistent there must be a constraint of disjointness between the explicitly declared names and those that occur within the instantiations. That is, in the scope of some $[\dagger X]$ a declaration like $X;\ a : T$ imposes a constraint that there is no occurrence of the name a within the schema X.

7 Projection

We can now define schema projection in a similar way to the above definitions, with the similar proviso that the name Y must be guaranteed not to be in the scope of the declarations introduced by the schema inclusions $S;\ T$.

$$\_ \upharpoonright \_ [\dagger X, Y] == \lambda S : \mathbb{P} X;\ T : \mathbb{P} Y \bullet \{S;\ T \bullet \theta Y\}$$

8 Further Schema Operations - Natural Composition

The schema calculus operations given in [5, 7] include three operations of schema quantification. For any two schemas X and Y we can form the expressions $\exists X \bullet Y$, $\forall X \bullet Y$ and $\exists_1 X \bullet Y$. [2] gives the first two of these, but omits $\exists_1$.

With the extensions to notation proposed here, we could define functions which restate these operations with different syntax, and can also define new functions using them in combination. The definitions of these operations in [5] require that in the case of $\exists X \bullet Y$, for example, all names in the signature of X must also be present in the signature of Y, and similarly for the other two. In [7] this restriction is relaxed, and the only constraint between the signatures is that they should be type-compatible. For further discussion and motivation of this point see [10]. We continue on the basis of the definitions in [7].

The operations $X \wedge Y$, $X \vee Y$, $X \Rightarrow Y$ and $X \Leftrightarrow Y$ all produce schema results whose signatures are formed by merging the signatures of the constituent schemas. The schema expression $\exists X \bullet Y$, however, has a value equal to that of $X \wedge Y$ followed by a hiding of those names present in the signature of X. Similarly, the schema expression $\forall X \bullet Y$ has a value equal to that of $X \Rightarrow Y$ followed by a hiding of those names present in the signature of X.

To illustrate the potential of these operations, and to prepare for the definitions of schema composition and piping below, we next define an operation which we could call "schema inner product" or "natural composition". We shall proceed with the latter name, and define

$$\begin{array}{l} \mathit{NatCompose}[\dagger X, Y] == \\ \quad \lambda S : \mathbb{P} X;\ T : \mathbb{P} Y \bullet \\ \quad\quad \{S;\ T \bullet \theta[(\exists Y \bullet X);\ (\exists X \bullet Y)]\} \end{array}$$

This function takes two schema arguments and produces a schema obtained by conjoining them, then hiding the common components. The type of the result is

$\mathbb{P}[(\exists\, Y \bullet X);\ (\exists\, X \bullet Y)]$, which can only be type-correct if $X \approx Y$. We observe that this operation is commutative, but is not in general associative.

We generalise this definition below to the cases of schema sequential composition and schema piping. In those cases we manipulate decorations on names, but these manipulations create the possibility of accidental coincidences between other names than those explicitly under consideration. It is then necessary to use a somewhat more complicated definition, equivalent to *NatCompose*, namely

$$
\begin{array}{l}
NatCompose2[\dagger X, Y] == \\
\quad \lambda\, S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet \\
\qquad \{b : [(\exists\, Y \bullet X);\ (\exists\, X \bullet Y)] \mid \\
\qquad\quad \exists\, m : \exists\exists\, Y \bullet \{b\} \bullet S \bullet \\
\qquad\quad m \in \exists\exists\, X \bullet \{b\} \bullet T\}
\end{array}
$$

where the values of b are the bindings in the result, and the values of m are the matching bindings, which are hidden. The value of the comprehension is formed as follows:
a) take some binding b of the result type;
b) $\{b\}$ is the schema whose sole member is that binding;
c) $\exists\, Y \bullet \{b\}$ is the schema whose sole member is a binding consisting of the components of b drawn from X;
d) $\exists\exists\, Y \bullet \{b\} \bullet S$ is the schema consisting of those bindings of the matching type that are consistent with S and the parts of b drawn from X;
e) similarly, $\exists\exists\, X \bullet \{b\} \bullet T$ is the schema consisting of those bindings of the matching type that are consistent with T and the parts of b drawn from Y;
f) the value b is included in the comprehension if there is a value of m that is a member of both these two schemas.

9 Heterogeneous State Transitions

The relaxation of the rules of generic typing given above has allowed us to define in Z six of the operations of schema calculus. In all of these cases no special recognition is given to the "decoration" of names. Other operations of the schema calculus, however, are designed to work with the "state and operations" convention, and to do so they treat different schema components differently according to the decorations attached to their names.

The descriptions of that convention [5, 2] are in terms of a single state schema, together with operations which relate the values of the components of that state schema with another schema resembling it but with its component names systematically dashed. Thus the components names might be a, b, c, a', b', c', where each of the pairs a, a', b, b', c, c' is declared as of the same type. Inputs, decorated with ?, and outputs, decorated with !, may also be present. We can describe this as the assumption of "homogeneous" state transitions, in that the type of dashed state differs from that of the undashed state only in the systematic dashing of the component names.

The operations of the schema calculus which are designed to work within the convention, namely precondition, schema composition, schema override (given in [2] only), and schema piping, make a weaker assumption, however, in that they assume that there is an undashed and a dashed state, but without any assumption that these resemble each other. That is, the recognition of a component as "undashed" is in no way dependent on the presence of a corresponding "dashed" component in the same schema, nor vice-versa. For example the components names might be a, b, c, x', y', z', where the components a, b, c will be recognised as undashed, and the components x, y, z recognised as dashed. In our formal description here we therefore treat these as heterogeneous (that is, not necessarily homogeneous) state transitions operations, and examine the restriction to homogeneity later.

10 Removing Decorations - Schema Precondition

In order to proceed further, we propose the introduction of an operator to "undecorate" any schema, irrespective of whether it is a generic parameter or not. For any schema S and decoration d the meaning of the expression

$\quad \text{undecor}\ \ d\ S$

is the schema S with all components without the decoration d hidden, and with the decoration d removed from all those component names that have it. For example, the value of

$\quad \text{undecor}\ \ '\ [a, b, b', c'' : \mathbb{N} \mid a = 3 \land b = 4 \land b' > a \land c'' > b]$

is equal to

$\quad [b, c' : \mathbb{N} \mid \exists\, a, b : \mathbb{N};\ b' == b;\ c'' == c' \bullet a = 3 \land b = 4 \land b' > a \land c'' > b]$

which in turn can be simplified to

$\quad [b, c' : \mathbb{N} \mid b > 3 \land c' > 4]$

This description makes the operation well-defined for all schemas and decorations. If the schema has no uses of the given decoration, the result is a schema of empty signature, either [ ] or [| false] (as legitimised fully in [7]).
For any schema S and decoration d, the equation

$\quad (\text{undecor}\ \ d\ (S\ d)) = S$

will always be valid, since the decoration is applied uniformly to all components, and then can be equally uniformly removed from them.
On the other hand the expression

$\quad (\text{undecor}\ \ d\ S)d$

is obtained by removing the decoration from those components which have it, hiding the others, then replacing the decoration on the result. This has the effect of hiding the components which do not have the decoration, and leaving the rest unchanged. We can use this to obtain the schema in which all components of a particular decoration have been hidden, by writing

$\quad \exists(\text{undecor}\ \ d\ S)d \bullet S.$

Using this we can define schema precondition, as

$$\text{pre}[\dagger X] == \lambda\, S : \mathbb{P}\, X \bullet \exists(\text{undecor}\ \ '\ X)';\ (\text{undecor}\ \ !\ X)! \bullet S$$

11 Schema Override

We can take schema override nearly verbatim from [2] and say

$$\_ \oplus \_ [\dagger X, Y] == \lambda S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet (S \wedge \neg\, (\mathrm{pre}\ T) \vee T)$$

and we have some nice theorems such as

$$[\dagger X, Y] \models ?$$
$$\forall S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet \mathrm{pre}(S \oplus T) = (\mathrm{pre}\, S \vee \mathrm{pre}\ T)$$

$$[\dagger X, Y, Z] \models ?$$
$$\forall S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y;\ U : \mathbb{P}\, Z \bullet (S \oplus T) \oplus U = S \oplus (T \oplus U)$$

12 Turning an Operation Schema into a Relation

Given a conventional operation schema, it is sometimes convenient to derive the corresponding relation, as needed by [6] for example.

$$\begin{array}{l} \mathit{relate}[\dagger X] == \\ \quad \lambda\, A : \mathbb{P}\, X \bullet \mathbf{let} \\ \qquad\qquad \mathit{in} == (\mathrm{undecor}\ ?\ X)?; \\ \qquad\qquad \mathit{out} == (\mathrm{undecor}\ !\ X)!; \\ \qquad\qquad \mathit{dashed} == (\mathrm{undecor}\ '\ X)' \bullet \\ \qquad \{i : \mathit{in};\ o : \mathit{out};\ d : \mathit{dashed};\ u : \exists\, \mathit{in};\ \mathit{out};\ \mathit{dashed} \bullet X \mid \\ \qquad\quad \forall\, bn : [\{i\};\ \{o\};\ \{d\};\ \{u\}] \bullet bn \in A \bullet \\ \qquad\qquad\qquad ((u, i), (d, o))\} \end{array}$$

13 Schema Composition

To define schema sequential composition we need to discover all dashed components of the first schema operand that match undashed components in the second. At the same time, we must not impose any unnecessary constraints; for example, the two operands need not even be compatible. The form of the definition is modelled on the second form of "natural composition" above, as:

$$\begin{array}{l} \_ \fatsemi \_ [\dagger X, Y] == \\ \quad \lambda\, S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet \\ \qquad \{b : [(\exists\ Y\ ' \bullet X);\ (\exists\, \mathrm{undecor}\ '\ X \bullet Y)] \mid \\ \qquad\quad \exists\, m : (\mathrm{undecor}\ '\ (\exists \exists\ Y\ ' \bullet \{b\} \bullet S)) \bullet \\ \qquad\quad m \in (\exists \exists\, \mathrm{undecor}\ '\ X \bullet \{b\} \bullet T)\} \end{array}$$

The definitions of schema composition given in [5, 2, 7] all include a further constraint that the base names that match should themselves be undecorated, that is, that the names in the binding m above should have no decoration. There seems to be no reason to impose that constraint, however, and therefore we do not propose the extensions to notation which doing so would require.

14 Type-correctness and Associativity of Schema Composition

The type of the result of schema sequential composition between two schemas of types $\mathbb{P}\, X$ and $\mathbb{P}\, Y$ respectively is

$\mathbb{P}[(\exists\, Y\,' \bullet X);\ (\exists\, \mathrm{undecor}\ '\ X \bullet Y)]$

provided that expression is type-correct. This is the case provided

$(\exists\, Y\,' \bullet X) \approx (\exists\, \mathrm{undecor}\ '\ X \bullet Y)$

which in turn requires that

$Y\,' \approx X$

and that

$\mathrm{undecor}\ '\ X \approx Y$

If these conditions are not met, the composition cannot be formed.

The operation of sequential composition is not in general associative, but it is associative whenever the formal statement of associativity is type-correct. That is,

$$[\dagger X, Y, Z] \models ?\ \forall\, S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y;\ U : \mathbb{P}\, Z \bullet (S \fatsemi T) \fatsemi U = S \fatsemi (T \fatsemi U)$$

is valid whenever it is well-typed. Explicitly:

$[(\exists[(\exists\, Z\,' \bullet Y);\ (\exists\, \mathrm{undecor}\ '\ Y \bullet Z)]\,' \bullet X);$
$\quad (\exists\, \mathrm{undecor}\ '\ X \bullet [(\exists\, Z\,' \bullet Y);\ (\exists\, \mathrm{undecor}\ '\ Y \bullet Z)])]$

must be the same type as

$[(\exists\, Z\,' \bullet [(\exists\, Y\,' \bullet X);\ (\exists\, \mathrm{undecor}\ '\ X \bullet Y)]);$
$\quad (\exists\, \mathrm{undecor}\ '\ [(\exists\, Y\,' \bullet X);\ (\exists\, \mathrm{undecor}\ '\ X \bullet Y)] \bullet Z)]$

More illuminating, however, is to note the counter-examples to associativity. The cases where the same three schemas can be composed together sequentially, with association to the left or to the right, so that either composition is type-correct but the two cases give different resultant types, are shown by the following three representative examples:

$[x'' : A] \fatsemi [x' : A] \fatsemi [x : A]$,
$[x' : A] \fatsemi [x' : A] \fatsemi [x : A]$,
$[x' : A] \fatsemi [x : A] \fatsemi [x : A]$.

15 Schema Piping

The definition of schema piping may be expressed thus:

$$\begin{array}{l} \_ \gg \_\ [\dagger X, Y] == \\ \quad \lambda\, S : \mathbb{P}\, X;\ T : \mathbb{P}\, Y \bullet \\ \quad\quad \{b : [(\exists(\mathrm{undecor}\ ?\ Y)! \bullet X);\ (\exists(\mathrm{undecor}\ !\ X)? \bullet Y)] \mid \\ \quad\quad\quad \exists\, m : (\mathrm{undecor}\ !\ (\exists\, \exists(\mathrm{undecor}\ ?\ Y)! \bullet \{b\} \bullet S)) \bullet \\ \quad\quad\quad m \in (\mathrm{undecor}\ ?\ (\exists\, \exists(\mathrm{undecor}\ !\ X)? \bullet \{b\} \bullet T))\} \end{array}$$

As with schema sequential composition above, this operation is associative whenever the alternative associations are of the same type, and the representative

counter-examples are:
$[x! : A] \gg [x! : A] \gg [x? : A]$
$[x! : A] \gg [x? : A] \gg [x? : A]$.

16 Rules of Refinement

The standard refinement rules can be stated formally using this notation. We follow [5] page 138 with minor modifications.

We do not explicitly define the abstract state or the concrete state. The schemas we use are:
a) an abstract operation schema Aop, whose components include those of the undashed and dashed abstract states, together with inputs and outputs as required;
b) a concrete operation schema Cop, whose components include those of the undashed and dashed concrete states, together with inputs and outputs as required;
c) an undashed abstraction schema Abs, which relates undashed concrete and abstract states, and possibly also their inputs;
c) a dashed abstraction schema $Abs2$, which relates dashed concrete and abstract states, and possibly also their outputs.

We can then give the refinement relation as

$$
\begin{array}{l}
refines\ [\dagger A, C, X, Y] == \\
\quad \{Aop : \mathbb{P}\, A;\ Cop : \mathbb{P}\, C;\ Abs : \mathbb{P}\, X;\ Abs2 : \mathbb{P}\, Y \mid \\
\qquad (\exists \,\mathrm{pre}\, A;\ \mathrm{pre}\, C \bullet X) \land \\
\qquad (\exists (\exists \,\mathrm{pre}\, A \bullet A);\ (\exists \,\mathrm{pre}\, C \bullet C) \bullet Y) \land \\
\qquad (\forall \,\mathrm{pre}\, Aop;\ Abs \bullet \mathrm{pre}\, Cop) \land \\
\qquad (\forall \,\mathrm{pre}\, Aop;\ Abs;\ Cop \bullet \exists (\exists \,\mathrm{pre}\, A \bullet A) \bullet Abs2 \land Aop)\}
\end{array}
$$

where the expression $(\exists \,\mathrm{pre}\, A \bullet A)$ is used to declare just the after states and outputs of Aop, and similarly the expression $(\exists \,\mathrm{pre}\, C \bullet C)$ is used to declare just the after states and outputs of Cop.
This differs from [5] in the following respects.
a) All the schemas have been declared. The necessary constraints on their components and their types arise from the requirement that the formal definition be well typed.
b) The main part of the formulation has been made shorter by using the operation and abstraction schemas directly as declarations, rather than as predicates.
c) The formulation is more general in allowing any number and names of inputs and of outputs.
d) Separate abstraction schemas Abs and $Abs2$ are used where [5] has Abs and $Abs\,'$. This allows the operation schemas to be heterogeneous in the sense explained above, and also allows refinement of the input and output components. The restriction to the cases considered by [5] is obtained by writing $Abs\,'$ for $Abs2$.

e) If any component is unchanged by refinement, it can be omitted from the abstraction schemas.
f) The concrete form is allowed to have extra after-states and outputs unrepresented in the abstract state, which will still leave (pre Cop) well-typed in the environment of (pre Aop; Abs). This extension arises naturally from the formalism, and accords with the practical realities of refinement.

Following [5] page 140, we can specialise to functional refinement, giving

$$
\begin{array}{l}
\mathit{frefines}\ [\dagger A, C, X, Y] == \\
\quad \{ Aop : \mathbb{P}\, A;\ Cop : \mathbb{P}\, C;\ Abs : \mathbb{P}\, X;\ Abs2 : \mathbb{P}\, Y \mid \\
\qquad (\exists\, \mathrm{pre}\, A;\ \mathrm{pre}\, C \bullet X) \land \\
\qquad (\exists (\exists\, \mathrm{pre}\, A \bullet A);\ (\exists\, \mathrm{pre}\, C \bullet C) \bullet Y) \land \\
\qquad (\forall\, \mathrm{pre}\, Aop;\ Abs \bullet \mathrm{pre}\, Cop) \land \\
\qquad (\forall\, \mathrm{pre}\, Aop;\ Abs;\ Cop;\ Abs2 \bullet Aop) \}
\end{array}
$$

and then operational refinement is derived as the case where Abs and $Abs2$ are empty, and so can be dropped, giving the form as in [5] page 136:

$$
\begin{array}{l}
\_\ \mathit{opRefines}\ \_\ [\dagger A, C] == \\
\quad \{ Aop : \mathbb{P}\, A;\ Cop : \mathbb{P}\, C \mid \\
\qquad (\forall\, \mathrm{pre}\, Aop \bullet \mathrm{pre}\, Cop) \land \\
\qquad (\forall\, \mathrm{pre}\, Aop;\ Cop \bullet Aop) \}
\end{array}
$$

17 Homogeneous State Transitions - Δ and Ξ

As stated above, the descriptions of the "state and operations" convention in [5, 2] are all in terms of homogeneous state transitions, in that the type of the dashed state differs from that of the undashed state only in the systematic dashing of the component names. Inputs, decorated with ?, and outputs, decorated with !, may also be present.

This approach makes extensive use of the convention whereby special meaning is given to schemas with names whose initial characters are Δ or Ξ. These characters are not themselves operators, but it has often been suggested that they should be. This would allow them to be applied to any schema-valued expression.
To define Δ we could say:

$$\Delta\ [\dagger X] == \lambda\, S : \mathbb{P}\, X \bullet [S;\ S\,']$$

which requires that S and $S\,'$ should be compatible schemas. The description in [5] requires that the base names should themselves be undecorated, but this seems an unnecessary restriction.
The operator Ξ would be defined similarly, as

$$\Xi\ [\dagger X] == \lambda\, S : \mathbb{P}\, X \bullet [S;\ S\,' \mid \theta X = \theta X\,']$$

These definitions could replace, or perhaps just supplement, the existing conventions.

18 Homogeneous Operation Schemas - Recognising the State

In order to define functions which take homogeneous state transition schemas as arguments, it is necessary to identify the state components. In the heterogeneous case, such as in the definition of "pre" above, we treat all components whose names do not have a dash as "undashed", and all components whose names have a dash as "dashed". For the homogeneous case, however, we only recognise as "undashed" those components for which there is a dashed counterpart, and vice versa. Other components may be present, but are treated as constant.

If we have some generic parameter X, constrained to be a schema and which we wish to treat as the type of a homogeneous operation schema, the expression that is equal to X after hiding of all dashed components that have corresponding undashed components is given by

$(\exists X' \bullet X)$

which leaves the undashed components together with the constant components. The expression

$(\exists(\exists X' \bullet X) \bullet X)$

therefore yields the type of the dashed components alone.

Similarly the expression that is equal to X after hiding of all undashed components that have corresponding dashed components is given by

$(\exists \operatorname{undecor}\ '\ X \bullet X)$

which leaves the dashed components together with the constant components. The expression

$(\exists(\exists \operatorname{undecor}\ '\ X \bullet X) \bullet X)$

therefore yields the type of the undashed components alone.

For these expressions to be type-correct, it is necessary that the type of each undashed component is the same as the type of its dashed counterpart.

19 Schema Iteration

To illustrate the possibilities opened up by the notation developed here, we define two forms of schema iteration. Neither of these appears as part of the standard schema calculus, but the facility has been called for in various forms, for example in [3].

For simplicity, we assume that all components of the schema to be iterated form matching pairs of undashed and dashed names of the same type, and that there are no "constant" components. We can than define a schema version of the existing toolkit function *iter*

$$
\begin{array}{l}
schIter[\dagger X] == \\
\quad \lambda\, n : \mathbb{N} \bullet \lambda\, S : \mathbb{P}\, X \bullet \\
\quad\quad \textbf{let}\ state == (\exists X' \bullet X) \bullet \\
\quad\quad\quad [X \mid (\theta state, \theta state') \in iter\ n\ \{S \bullet (\theta state, \theta state')\}]
\end{array}
$$

We give also a schema version of a "while" loop. This is based on the function *do* as described in [9], which can be defined as

$$do[X] == \lambda R : X \leftrightarrow X \bullet \bigcap \{ Q : X \leftrightarrow X \mid \mathrm{id}(X \setminus \mathrm{dom}\, R) \subseteq Q \land R \fatsemi Q \subseteq Q \}$$

For any function f the effect of applying $do\ f$ to an argument is the same as the effect of repeatedly applying f to that argument until the result is no longer in the domain of f. Similarly for any relation R the effect of taking the relational image of $do\ R$ through a set is the union of all values obtained by taking the relational image of R through elements of that set until the result is no longer in the domain of R.
Using this we define the schema version:

$$schDo[\dagger X] == \lambda S : \mathbb{P}\, X \bullet \mathbf{let}\ state == (\exists X' \bullet X) \bullet [X \mid (\theta state, \theta state') \in do\ \{S \bullet (\theta state, \theta state')\}]$$

The effect of applying *schDo* S to some state is that of repeatedly re-applying S to that state until the precondition is not satisfied.

20 Implementation

We have implemented these proposals in the CADiℤ tool. The implementation of "undecor" proved to be straightforward. The main proposal, to allow the introduction of generics constrained to be schemas, was less simple but we believe we have solved it successfully. The details are given in a companion paper [8].

21 Conclusions

We have proposed two extensions to Z. The first is the operation "undecor" which allows for the explicit removal of decorations from a schema. The second is a relaxation of restriction whereby generic parameters are allowed to have their types partially constrained. We have shown how this makes it possible to define in Z all the complex schema calculus operations such as sequential composition and piping.

22 Acknowledgments

This work was done as part of the project "Standardising Z Semantics", for which Dr. King is principal investigator, and Mr. Valentine and Dr. Toyn are receiving EPSRC funding (Grant number GR/M 20723).

References

1. J. G. Hall and A. P. Martin. W reconstructed. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM '97: The Z Formal Specification Notation*, LNCS 1212, Reading, April 1997. Springer.
2. I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall, second edition, 1993.
3. P. Luigi Iachini. Operation schema iterations. In J. E. Nicholls, editor, *Z User Workshop*, Oxford, December 1990. Springer.
4. Andrew Martin. A revised deductive system for Z. Technical Report 98 - 21, Software Verification Research Centre, University of Queensland, 1998.
5. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
6. Susan Stepney and David Cooper. Formal methods for industrial applications. (These proceedings), 2000.
7. I. Toyn, editor. *Z Notation*. ISO, 1999. Final Committee Draft, available at http://www.cs.york.ac.uk/~ian/zstan/.
8. Ian Toyn, S. H. Valentine, Susan Stepney, and Steve King. Typechecking Z. (These proceedings), 2000.
9. S. H. Valentine. Z – –, an executable subset of Z. In J. E. Nicholls, editor, *Z User Workshop*, pages 157–187, York, December 1991. Springer.
10. S. H. Valentine. Equal rights for schemas in Z. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95*, LNCS 967, pages 183–202, Limerick, September 1995. Springer.
11. J. C. P. Woodcock and S. M. Brien. W: a logic for Z. In J. E. Nicholls, editor, *Z User Workshop*, York, December 1991. Springer.