

# Modular refinement in novel schema calculi

Moshe Deutsch, Martin C. Henson and Besnik Kajtazi  
Department of Computer Science, University of Essex, UK.  
{mdeuts, hensm, bkajta}@essex.ac.uk

## Abstract

Using the language  $Z$  for more than specification is hindered by the fact that its algebra of schemas is not monotonic with respect to refinement; so specification is modular, but development is not. In this paper we isolate the reasons why  $Z$  suffers from these problems and we describe alternative models for specifications and schema operations which are monotonic. This leads us to explore a number of theoretical and pragmatic questions: the former concern logics for modular refinement; the latter explore the use of novel schema calculi in practice.

## 1 Introduction

$Z$  [1] is a specification language which permits the modular construction of specifications by means of an algebra of connectives and quantifiers reminiscent of predicate logic. This provides great structural expressivity and is a major reason for the popularity of the approach. Practical examples are covered in the better textbooks (e.g. [3], [18]) and its logical properties, justifying the usual terminology: schema *calculus*, in [13].

In addition to its use purely as a language for specification, there has been considerable interest in using  $Z$  for design and development. An approach which integrates work in data refinement [11] with  $Z$  is given in [18] and, in program development, there are approaches which seek to combine the refinement calculus (e.g. [16]) with  $Z$  (e.g. [14] and [5, 6]).

A number of issues make these more general uses for  $Z$  problematic, the most central being the fact that the modular techniques for expressing structured specifications is not accompanied by the possibility of modular *reasoning* because the schema operations are not monotonic with respect to the standard account of refinement.

There is one approach to program development in the schema calculus which takes a more radical step: it modifies the underlying semantics of schemas and thereby establishes a monotonic  $Z$ -like schema calculus [12]. The precise

relationship between this and the standard approach is, however, unclear.

This paper aims to shed some light on the general issue of modular reasoning in a schema calculus for specification. It builds on earlier work on  $Z$  logic, on refinement, and on monotonicity properties for specification operations. Much of the previous related work is technical in nature, whereas this paper has an ambition in the main to scene and agenda setting and review. One of our objectives here is to piece together some of the themes emerging from the literature and to propose some avenues for further exploration: after all, successful integration of modular reasoning with modular specification addresses the “scaling-up” problem – possibly the most important challenge in this area of formal methods.

We begin with an overview of the limitations of  $Z$  regarding modular reasoning. We do so not so much to highlight its deficiencies as to stress its strengths: it permits a combination of highly abstract description with a very flexible and expressive framework for structuring and organising in the large. We then provide two contrasting alternative models for schemas. The first interprets an operation schema as a set of possible implementations and the second is an alternative relational model based on a relative of the lifted-totalisation to be found in, for example, [18].

Some familiarity with  $Z$  and the standard account of refinement at textbook level (e.g. [18]) is assumed, but not an acquaintance with  $Z$ -logic and its associated theory. Here we have aimed to present the technical material relatively informally; everything we describe in the paper can be cashed out in detail by the interested reader through consultation of the relevant references to the literature.

## 2 $Z$ and modular reasoning

### 2.1 Modular description in $Z$

As an example of modular description in  $Z$ , consider the *promotion* of a local operation  $LocalOp$  over a local state  $LocalState$  to a global operation  $GlobalOp$  over a global state  $GlobalState$  by means of a specification  $\Phi Link$  ex-

plaining how the local state is integrated in the global state:

$$GlobalOp \hat{=} \exists A LocalState \bullet LocalOp \wedge \Phi Link$$

Here, conjunction is used to connect the local operation to the global state and the existential quantifier *hides* the local state in the global operation.

This is a standard specification technique of long standing. There are plenty of interesting examples in the textbooks, and its fine structure has recently been characterised as a specification *pattern* in [17].

In the context of a Z logic, it is possible to reason about this, and other, specifications. But there is a certain key relation, *refinement*, which does not interact well with the various schema operators. In particular, one would hope that an ability to write modular specifications would be accompanied by a similar ability to undertake modular reasoning. This would require the schema operations to be *monotonic* with respect to the key relation of refinement. In particular, if *NewLocalOp* is a refinement of *LocalOp*, that is:

$$NewLocalOp \sqsupseteq LocalOp$$

then we would like *NewGlobalOp*, defined:

$$NewGlobalOp \hat{=} \exists A LocalState \bullet NewLocalOp \wedge \Phi Link$$

to satisfy:

$$NewGlobalOp \sqsupseteq GlobalOp$$

In general this does not follow in Z.

## 2.2 Partial relation semantics and equational logic

In Z, schemas generally denote *partial* relations. For example, consider the schema:

$\begin{array}{l} \text{Predecessor} \\ \hline x, x' : \mathbb{N} \\ \hline x' = x - 1 \\ \hline \end{array}$
---

The relation this denotes contains *bindings* (valid assignments of values to the observations  $x$  and  $x'$ ) of the form:

$$\langle x \Rightarrow n, x' \Rightarrow n - 1 \rangle$$

for all  $n > 0$ . But no binding of the form:

$$\langle x \Rightarrow 0, x' \Rightarrow m \rangle$$

for any  $m \in \mathbb{N}$ . In this sense the schema is partial when viewed as a relation between its *before* observation  $x$  and *after* observation  $x'$ . We will use  $U$  (etc.) in future to range over such relations.

In fact the underlying relation of Z is not refinement but equality. The definition of the schema operators leads to

an equational logic. One such equation, characterising conjunction, is:

$$[D_0 \mid P_0] \wedge [D_1 \mid P_1] = [D_0; D_1 \mid P_0 \wedge P_1]$$

There are similar equations for all schema operators. By orienting these equalities, it is easy to see that *every* schema expression is trivially equal to a single atomic schema (its normal form so to speak).

Note that the equational logic is a property of the partial relation semantics (see [13] for technical details), hence one must construe equality in terms of subset and then it would be possible to take refinement to be the fundamental relation by means of:

$$U_0 \sqsupseteq U_1 =_{df} U_0 \subseteq U_1$$

But this is only a partial correctness interpretation in which preconditions cannot be weakened.<sup>1</sup> This has not been found to be satisfactory in practice but does have the benefit of leading to a fully monotonic schema calculus.

## 2.3 Total relation semantics and refinement

The standard interpretation of refinement for Z is:

$$U_0 \sqsupseteq U_1 =_{df} \dot{U}_0 \subseteq \dot{U}_1$$

where  $\dot{U}$  is the *lifted totalisation* of  $U$ . Let  $\mathbb{N}_\perp =_{df} \mathbb{N} \cup \{\perp\}$  then, in addition to the values discussed above, *Predecessor* also contains:

$$\langle x \Rightarrow 0, x' \Rightarrow m \rangle$$

and

$$\langle x \Rightarrow \perp, x' \Rightarrow m \rangle$$

for all  $m \in \mathbb{N}_\perp$ .

It is important to note that this definition concerns the partial relation interpretation of schema *expressions*. That is, the interpretation of schemas, and of the operations for building modular specifications, are logically *prior* to the theory refinement.

This account has several desirable features: it is a total correctness interpretation, postconditions can be strengthened and preconditions can be weakened. It has flaws: one arguably minor and one major.

**The minor flaw** Consider the following schemas:

$\begin{array}{l} A_0 \\ \hline x, x' : \mathbb{N} \\ \hline x' = 0 \\ \hline \end{array}$
--

<sup>1</sup> Z interprets partiality as under-definition but there is no reason why the partiality could not be interpreted as over-definition, that is a magical rather than chaotic (or abortive) interpretation. See below.

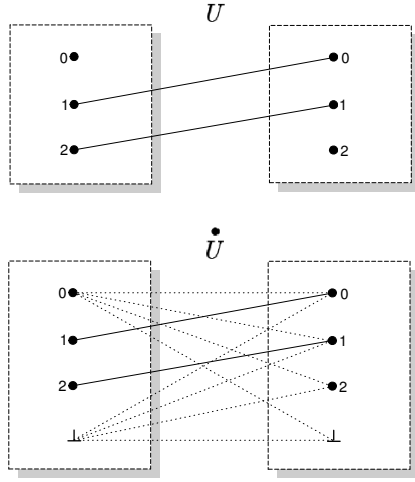


Fig. 1. The lifted totalisation illustrated.

and:

$$\frac{A_1}{\begin{array}{c} x, x' : \mathbb{N} \\ x' = 1 \end{array}}$$

Their conjunction  $A_0 \wedge A_1$ , given the equational logic, is equivalent to:

$$\frac{A}{\begin{array}{c} x, x' : \mathbb{N} \\ x' = 0 \wedge x' = 1 \end{array}}$$

The model of this is nowhere defined. Recall that, as mentioned above, Z interprets partiality chaotically, as under-specification. This cashes out once we also recall that refinement applies to expressions. Thus we may refine A by exhibiting a subset of  $\dot{A}$ . For example, the following will do:

$$\frac{B}{\begin{array}{c} x, x' : \mathbb{N} \\ x' = 9 \end{array}}$$

That is, we have:

$$B \supseteq A$$

This seems counterintuitive, but only because we may be inclined to assume that partiality is a manifestation of magic rather than chaos.

**The major flaw** The major flaw is that *no schema operator is monotonic with respect to this theory of refinement*. One way of illustrating this failure is to take a look at how the lifted-totalisation interacts with the schema operators. For example, if the following full distributivity property held:

$$U_0 \dot{\wedge} U_1 = \dot{U}_0 \wedge \dot{U}_1 \quad \times$$

then schema conjunction *would* be monotonic with respect to refinement. That is, we would have:

$$\frac{U_0 \supseteq U_2 \quad U_1 \supseteq U_3}{U_0 \wedge U_1 \supseteq U_2 \wedge U_3} \quad \times$$

with proof:

$$\frac{\frac{\frac{1}{z \in (U_0 \wedge U_1)}}{z \in \dot{U}_0 \wedge \dot{U}_1} \quad \times \quad \frac{\frac{1}{z \in (U_0 \dot{\wedge} U_1)}}{z \in \dot{U}_0 \wedge \dot{U}_1} \quad \times}{\frac{U_0 \supseteq U_2 \quad z \in \dot{U}_0}{z \in \dot{U}_2} \quad \frac{U_1 \supseteq U_3 \quad z \in \dot{U}_1}{z \in \dot{U}_3}}{\frac{z \in \dot{U}_2 \wedge \dot{U}_3}{z \in (U_2 \dot{\wedge} U_3)} \quad \checkmark} \quad 1$$

Here, the annotations indicate the problem: only half of the full distributivity equation holds. Put another way: we needed the equation at the level of the total relation semantics, but we only have it at the level of the partial relation semantics.

A similar situation arises with every schema operation. For example, these distributivity inequations hold (and none of the converses):

- (i)  $U_0 \dot{\circ} U_1 \subseteq \dot{U}_0 \circ \dot{U}_1$
- (ii)  $U_0 \dot{\vee} U_1 \subseteq \dot{U}_0 \vee \dot{U}_1$
- (iii)  $\dot{U}_0 \wedge \dot{U}_1 \subseteq U_0 \wedge U_1$
- (iv)  $\exists z : \dot{T} \bullet U \subseteq \exists z : T \bullet \dot{U}$

Of course, the problem may lie with the lifted-totalisation model for refinement. This is not so: that model is in some sense canonical. There are plenty of possible alternative approaches:

- *Proof-theoretic refinement* – it is possible to characterise refinement completely in terms of the basic considerations: preconditions may not strengthen and post-conditions may not weaken.
- *Weakest precondition refinement* – it is possible to reinterpret the partial relations in terms of a weakest precondition semantics and to characterise refinement in the standard way in that regime.

- *Sets of implementation* – in the spirit of constructive theories of program development, e.g. Martin-Löf Type Theory [15] (though in the setting of classical logic) it is possible to reinterpret specifications as sets of permissible implementations. Refinement in this case is simply set inclusion.
- *Strictly lifted totalisation* – it is possible to modify the lifted-totalisation so that the lifting is strict (abortive) rather than non-strict (chaotic).
- *Non-lifted totalisation* – it is possible to totalise the partial relations without lifting if one is prepared to exclude fully chaotic behaviour from the notion of precondition.

All five alternative theories of refinement described above are equivalent to the standard lifted-totalised account. As a consequence, all suffer from the same weaknesses in terms of their (lack of) monotonicity properties. The technical details are to be found in [7].

It is certainly worth asking under what circumstances full monotonicity is available in Z. This has been addressed in [10] for conjunction and disjunction, and for these, composition and the existential quantifier in [8]. The conditions are quite strong and, although they can be construed as *healthiness conditions*, are arguably too cumbersome to be useful in practice because they are proof-theoretically non-trivial, rather than simply syntactic in nature (c.f. for example, the side-conditions on a rule such as existential elimination in predicate logic).

To summarise: Z takes a *layered* approach to refinement. The underlying semantics of schema expressions is partial relational, but refinement is based on a subsequent interpretation. This may take many guises (including the standard, lifted-totalisation model) but all lead to equivalent theories for which the schema operators are not monotonic. Z does however have a (very reductive) equational logic.

This leaves us with an interesting question: *Which is more important: the equational logic or monotonicity of the schema operations?* – evidently we cannot have both simultaneously. Historically, the clear answer is *the equational logic*, though this may be because, until very recently, the lack of a mathematical analysis precluded addressing what was an unasked question. It is quite reasonable to take it that there is no definitive answer and that the matter is settled by features of the application context. It is also quite reasonable to explore the consequences of the alternative position: if we abandon the equational logic we may be able to rehabilitate monotonicity. There are a number of ways in which this might be done: in the remainder of the paper we will explore two such possibilities.<sup>2</sup> These are covered in the next

<sup>2</sup> Though motivated by entirely separate considerations (concerning software evolution) [9] is a careful working out of part of this project, where the underlying semantics is modified to a

two major sections of the paper. Before we turn to that we finish our analysis of Z by examining its notion of precondition.

## 2.4 Preconditions and postconditions

Z schemas are single-predicate: there are no separate syntactic preconditions. In this regard it differs from many other frameworks, like the specification statements of the refinement calculus [16] and the machine operations in the specification language B [2]. In Z, preconditions are logically induced as *feasibility conditions*; in general these will be stronger than their syntactic counterparts.

If one is content to use Z for specification and design then this postcondition approach has much to recommend it. On the other hand, if one wishes to develop programs through refinement, the logical precondition imposes a severe burden, in the worst case one which is *equivalent to deriving a program*.

To illustrate this, consider the following specification:

Sort
$x, x' : List \mathbb{N}$
$ordered(x')$ $permutation(x, x')$

The (logical) precondition is formed by the existential closure of the defining predicate with respect to the after observations. In this case:

$$\exists x' : List \mathbb{N} \bullet ordered(x') \wedge permutation(x, x')$$

But this is, essentially, the specification of sorting as a (an implicit)  $\Pi_2^0$  statement: the kind of specification routinely encountered in, for example, program derivation in type-theory (e.g. [4]). It heralds an *invariance of difficulty* in circumstances in which the precondition must be analysed (despite its being logically equivalent to *True*).

By contrast, in a framework in which syntactic preconditions are distinct from feasibility conditions, one would rather write:

Sort
$x, x' : List \mathbb{N}$
True
$ordered(x')$ $permutation(x, x')$

resulting in much greater logical simplicity in the program derivation.

weakest precondition model. At some point it will be necessary to compare this with the models introduced below.

Since our objective is to consider rational reconstructions of  $Z$  which are suitable for refinement to code, we shall proceed within a two-predicate framework. In addition to the benefits discussed here, we will see that the additional syntactic distinction leads to finer-grained notions of refinement, which are capable of distinguishing more extreme specifications, and yield a variety of refinement inequations which would not be possible otherwise.

### 3 A sets-of-implementations model

Since a parasitic theory of refinement, on built on top of an underlying partial relation semantics, cannot lead to a monotonic schema calculus, we will replace the semantics for schemas and schema expressions entirely. In this section we illustrate the approach taken in [12] and consider a *sets-of-implementations* model.

In this regime our mathematical model of a schema is the set of implementations which can be said to meet it. Refinement will then be straightforward: a refinement is a subset of implementations.

We define the implementation relation as follows:

$$f \in [D \mid P \mid Q] =_{df} \forall z^{\mathbb{W}} \bullet z.P \Rightarrow f(z).Q$$

where  $z.P$  is the assertion that  $P$  holds in the state (binding)  $z$  and  $\mathbb{W}$  is a *universal state* (a sufficiently large schema type) which contains, in particular,  $D$ .

Then the semantics of (atomic) schemas is immediate:

$$\llbracket U \rrbracket =_{df} \{f : \mathbb{W} \rightarrow \mathbb{W} \mid f \in U\}$$

As is refinement:

$$U_0 \sqsupseteq U_1 =_{df} \llbracket U_0 \rrbracket \subseteq \llbracket U_1 \rrbracket$$

But at this point the schemas in question are all atomic; we need to provide an interpretation for compound schema expressions by recursion on the language. For schema conjunction and disjunction we can simply take intersection and union of models:

$$\llbracket U_0 \wedge U_1 \rrbracket =_{df} \llbracket U_0 \rrbracket \cap \llbracket U_1 \rrbracket$$

and:

$$\llbracket U_0 \vee U_1 \rrbracket =_{df} \llbracket U_0 \rrbracket \cup \llbracket U_1 \rrbracket$$

Since the elements of the models are state transformations (over the global state  $\mathbb{W}$ ), we have a natural interpretation for schema composition:

$$\llbracket U_0 \circ U_1 \rrbracket =_{df} \{f_1 \circ f_0 \mid f_0 \in U_0 \wedge f_1 \in U_1\}$$

Finally, we consider hiding. The account we give (following [12]) hides a pair of complementary labels:

$$\llbracket \exists z, z' : C \bullet U \rrbracket =_{df} \{(\lambda \sigma \bullet g\sigma)[z/\sigma.z] \mid g \in \llbracket U \rrbracket\}$$

This rather inscrutable definition makes more intuitive sense in the context of a little program semantics:

$$\llbracket \text{begin var } z; \text{ cmd end} \rrbracket =_{df} (\lambda \sigma \bullet \llbracket \text{cmd} \rrbracket \sigma)[z/\sigma.z]$$

That is to say, the implementations of schemas with hidden observations are *blocks*.

This completes the interpretation. All four operators described are *fully monotonic* with respect to this notion of refinement. That is to say we have, for example:

$$\frac{U_0 \sqsupseteq U_1}{\exists z, z' : C \bullet U_0 \sqsupseteq \exists z, z' : C \bullet U_1}$$

and:

$$\frac{U_0 \sqsupseteq U_1 \quad U_2 \sqsupseteq U_3}{U_0 \circ U_2 \sqsupseteq U_1 \circ U_3}$$

As we discussed earlier, this interpretation cannot be standard and will not validate the usual equational logic. Instead we obtain an *inequational logic* of refinement. For example, we get these (among other) inequations for conjunction:

$$\begin{aligned} [T_0; T_1 \mid P_0 \vee P_1 \mid Q_0 \wedge Q_1] &\sqsupseteq [T_0 \mid P_0 \mid Q_0] \wedge [T_1 \mid P_1 \mid Q_1] \\ [T_0 \mid P_0 \mid Q_0] \wedge [T_1 \mid P_1 \mid Q_1] &\sqsupseteq [T_0; T_1 \mid P_0 \wedge P_1 \mid Q_0 \vee Q_1] \end{aligned}$$

For the hiding quantifier, we have:

$$\begin{aligned} \exists x, x' : T \bullet [x, x' : T; D \mid P \mid Q] &\sqsupseteq \\ [D \mid \exists u : T \bullet P[x/u] \mid \exists u, v : T \bullet Q[x, x'/u, v]] &\end{aligned}$$

And for composition we have this:

$$\begin{aligned} [x, x' : T \mid P_0 \mid Q_0] \circ [x, x' : T \mid P_1 \mid Q_1] &\sqsupseteq \\ [x, x' : T \mid P_0 \wedge \forall u : T \bullet Q_0[x'/u] \Rightarrow P_1[x/u] \mid & \\ \exists v : T \bullet Q_0[x'/v] \wedge Q_1[x/v]] &\end{aligned}$$

### 4 A total relational model

As an alternative approach we replace the usual partial relation semantics with the lifted-totalised interpretation. When  $U$  is an atomic schema we set:

$$\llbracket U \rrbracket =_{df} \dot{U}$$

Then the semantics is extended to schema operators more or less as in the standard account, for example:

$$\llbracket U_0 \wedge U_1 \rrbracket =_{df} \llbracket U_0 \rrbracket \cap \llbracket U_1 \rrbracket$$

Note, that here this is intersection of relations, rather than sets of functions, as it was in the previous section.

Refinement is now completely straightforward:

$$U_0 \sqsupseteq U_1 =_{df} \llbracket U_0 \rrbracket \subseteq \llbracket U_1 \rrbracket$$

And full monotonicity for all operators is immediate.

The various semi-distributivity results presented earlier become, under this model, an inequational logic. Translated, they become:

$$\begin{aligned} [D_0 \mid P_0] \wedge [D_1 \mid P_1] &\sqsupseteq [D_0; D_1 \mid P_0 \wedge P_1] \\ [D_0; D_1 \mid P_0 \vee P_1] &\sqsupseteq [D_0 \mid P_0] \vee [D_1 \mid P_1] \\ [D \mid \exists u : T \bullet P[z/u]] &\sqsupseteq \exists z : T \bullet [z : T; D \mid P] \end{aligned}$$

Alternatively, we may provide an interpretation of two-predicate schemas with the “lifted-totalised” interpretation for atomic schemas<sup>3</sup>

$$\llbracket [D \mid P \mid Q] \rrbracket =_{df} \{z_0 \star z'_1 \mid z_0.P \Rightarrow z_0.z'_1.Q\}$$

Here, we write  $z_0 \star z'_1$  for a typical binding, where  $z_0$  is the sub-binding concerning before observations, and  $z'_1$  is the sub-binding concerning after observations.

In fact, in view of the results of [7] we do not need non-strict lifting and we can use simpler relational models, established by:<sup>4</sup>

$$\llbracket [D \mid P \mid Q] \rrbracket =_{df} \{z_0 \star z'_1 \mid z_0.P \vee z_1 \neq \perp \Rightarrow z_0.z'_1.Q\}$$

Naturally, all the schema operators are monotonic with respect to refinement under either definition. It is worth illustrating just how immediate this is. Consider the following rule:

$$\frac{U_0 \wedge U_1}{U_0 \wedge U_2 \sqsupseteq U_1 \wedge U_2}$$

Its proof is:

$$\frac{\frac{U_0 \sqsupseteq U_1 \quad \frac{z \in U_0 \wedge U_2}{z \in U_0} 1}{z \in U_1} \quad \frac{z \in U_0 \wedge U_2}{z \in U_2} 1}{z \in U_1 \wedge U_2} 1$$

Here, for ease of presentation we have written  $z \in \llbracket U \rrbracket$  simply as  $z \in U$ .

The following refinement inequations for conjunction (among others) are derivable, in this interpretation:

$$\begin{aligned} [T_0 \mid P_0 \mid Q_0] \wedge [T_1 \mid P_1 \mid Q_1] &\sqsupseteq [T_0; T_1 \mid P_0 \wedge P_1 \mid Q_0 \vee Q_1] \\ [T_0; T_1 \mid P_0 \vee P_1 \mid Q_0 \wedge Q_1] &\sqsupseteq [T_0 \mid P_0 \mid Q_0] \wedge [T_1 \mid P_1 \mid Q_1] \end{aligned}$$

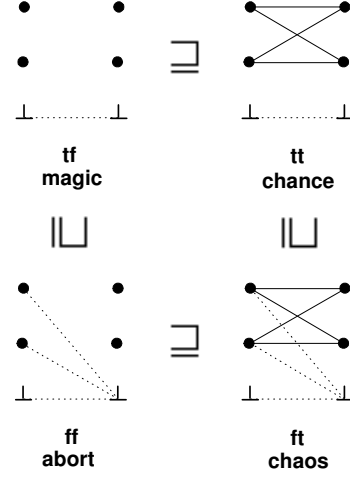
Note that these are the same as those given above for the sets-of-implementations interpretation.

One further benefit of syntactic preconditions is increased expressivity. This is illustrated by means of figure 2

<sup>3</sup> There is a reason for the scare-quotes here: beware that this does not in general create total relations. See figure 2 for example.

<sup>4</sup> In fact, this definition is reminiscent of the model with firing conditions as preconditions, in the single predicate regime: in the two-predicate regime the definition does not prevent weakening of preconditions.

below, which distinguishes four extreme specifications. Under the single predicate regime, in which the predicate and the induced feasibility condition must be true or false together, one can only express (versions of) *chance* and *abort*.



**Fig. 2. Extreme specifications.** The preconditions and postconditions take four possible combinations of *true* and *false*.

It is worth noting that, in this framework, partiality is more sensibly treated as magical, rather than chaotic, behaviour.

## 5 Extending the schema calculus

Up to this point we have considered alternative interpretations of existing schema operations, such as conjunction and disjunction; we have not considered alternative schema operations.

In attempting to integrate schema-based specification with design and implementation the focus changes: schema operations are not merely opportunities for structuring specifications, they become opportunities for structuring designs and ultimately implementations. This may suggest new operations for new purposes.

In this section we will consider just two: *schema abstraction* and *application*. These are introduced to interface with procedural abstraction and application.

The most well-developed approach is that based on the sets-of-implementations model which we described above, so we will begin with this and follow it with an ultimately simpler approach, in the relational framework. Our empha-

sis is conceptual and pragmatic: the interested reader is encouraged to consult [12] for full technical details.

## 5.1 schema abstraction and application

**A sets-of-implementation approach** The idea is to introduce a *schema valued function*. Suppose  $f \in \mathbb{W} \rightarrow \mathbb{W}$  satisfies  $f \in U$ , that is, it implements  $U$ . Then  $\text{curry}_{[z:T]} f$ , of type,  $T \rightarrow \mathbb{W} \rightarrow \mathbb{W}$ , is defined by:

$$\text{curry}_{[z:T]} f \ t^T \ \sigma =_{df} f \ \sigma[z/t]$$

This allows us to interpret our new operation:

$$\llbracket \lambda z : T \bullet U \rrbracket =_{df} \{ \text{curry}_{[z:T]} f \mid f \in \llbracket U \rrbracket \}$$

Note that this is *not* a new kind of schema: although its model is indeed a set of functions, they have the wrong type (they are not in  $\mathbb{W} \rightarrow \mathbb{W}$ ).

One obtains schemas from abstractions by application. Semantically this is fairly easy. For any schema abstraction  $\eta$  and term  $t$  of appropriate types:

$$\llbracket \eta[t] \rrbracket =_{df} \{ \lambda \sigma \bullet f(\sigma.t) \ \sigma \mid f \in \llbracket \eta \rrbracket \}$$

Syntactically, we proceed as follows. Let  $t$  be a term such that  $\alpha t$  and  $\alpha D$  are disjoint, and let  $D_0$  be the declaration corresponding to the observations of  $t$ . Then, syntactic application (substitution), for atomic schemas, is:

$$[D \mid P \mid Q][z/t] = [D; D_0 \mid P[z/t] \mid Q[z/t]]$$

It is then possible to prove that:

$$(\lambda z \bullet U)[t] = U[z/t]$$

**A relational approach** In this model, the denotation of a schema abstraction will be a *relation valued function*. Suppose that the before state of  $U$  is  $T^{in} \star [z : T]$  and the after state  $T^{out}$ , then the bindings of  $U$  will have type  $T^{in} \star [z : T] \star T^{out}$ . If we abstract with respect to the observation  $z$  then we will have a set of bindings of type  $T^{in} \star T^{out}$ , where the value of  $z$  is fixed by the abstraction. More exactly:

$$\llbracket \lambda z : T \bullet U \rrbracket =_{df} \lambda v \bullet \{ z_0 \star z'_1 \mid z_0 \star \langle z \Rightarrow v \rangle \star z'_1 \in \llbracket U \rrbracket \}$$

Again, as with the previous approach, this is not a schema: evidently it is not a relation, but rather a relation valued function. Similarly, one obtains a schema by application:

$$\llbracket \eta[t] \rrbracket =_{df} \llbracket \eta \rrbracket (\llbracket t \rrbracket)$$

Proving the  $\beta$ -equality:

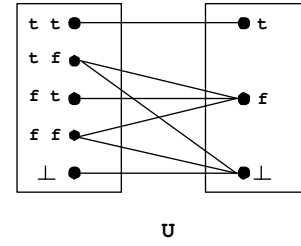
$$(\lambda z \bullet U)[t] = U[z/t]$$

is trivial in this case; we will illustrate it by example.

Consider the schema:

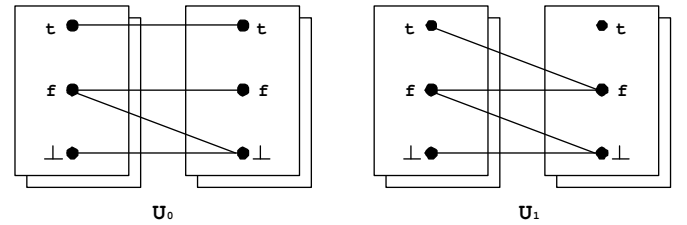
$U$
$x, x' : \mathbb{B}$
$z? : \mathbb{B}$
$x = \text{true}$
$x' = x \wedge z?$

The relational model of this is shown in figure 3 (the inputs are shown are  $z?$  and  $x$  respectively).



**Fig. 3.** The model of the specification  $U$ .

If we abstract with respect to the observation  $z?$  and apply the result to *true* or to *false* then we get the models shown in figure 4. These are the models of  $U_0 =_{df} (\lambda z? \bullet U) \text{true}$  and  $U_1 =_{df} (\lambda z? \bullet U) \text{false}$ .



**Fig. 4.** The models of the schema application expressions  $U_0$  and  $U_1$ .

Now consider the syntactic application of the values *true* and *false* to  $U$ . That is, the schemas  $U[z?/\text{true}]$  and  $U[z?/\text{false}]$ . These are:

$x, x' : \mathbb{B}$
$x = \text{true}$
$x' = x$

and:

$x, x' : \mathbb{B}$
$x = true$
$x' = false$

The relational model of these two schemas are those given in figure 4. This illustrates  $\beta$ -reduction in the relational model.

**Programming logic** In both the approaches sketched above, it is possible to develop programming logic rules which connect specification to code. For example, consider the following schema for recursive procedures over natural numbers:

`proc p[z] cases z in 0 : cmd0 | m + 1 : cmd1 endcases` where the term  $p[z]$  may occur in  $cmd_1$ . We can derive, for a suitable semantics (see, for example, [12] for the technical details and extensive proofs) a rule for *recursive procedure synthesis*:

$$\frac{cmd_0 \in U[z/0] \quad p[m] \in U[z/m] \vdash cmd_1 \in U[z/m+1]}{\text{proc } p[z] \text{ cases } z \text{ in } 0 : cmd_0 \mid m + 1 : cmd_1 \text{ endcases} \in U}$$

This is very intuitive and appealing. The rule holds in both models. We have considered the implementation relation in the sets-of-implementations model already; in the relational model it is redundant and essentially just a refinement  $f \sqsupseteq U$ . Note that the use of abstraction and application is implicit in the syntactic substitutions appearing in the two premises.

Similarly, rules can be derived for other programming features. For example, those we will need in the next section are as follows.

First for the skip command:

$$\frac{\text{skip} \in [D \mid P \mid Q] \quad \sigma.P}{\sigma.\sigma'.Q}$$

and:

$$\frac{\sigma.P \vdash \sigma.\sigma'.Q}{\text{skip} \in [D \mid P \mid Q]}$$

The following rule (in which no after state identifier may occur in the expressions  $exp_i$ ) is derivable for simultaneous assignment to the variables  $x_i$ :

$$\frac{\sigma.P \vdash \sigma.\sigma'[\dots x_i/\sigma.exp_i \dots].Q}{\dots x_i \dots := \dots exp_i \dots \in [\dots x_i, x'_i \dots : \mathbb{N}; D \mid P \mid Q]}$$

Command sequencing is straightforward:

$$\frac{cmd_0 \in U_0 \quad cmd_1 \in U_1}{cmd_0 ; cmd_1 \in U_0 ; U_1}$$

Note that this last rule is essentially a monotonicity result for schema composition.

## 6 Example derivation

We considering the simplest possible example.<sup>5</sup> Consider the following specification. The state is:

$$S =_{df} [x : \mathbb{N}]$$

The operation schema is:<sup>6</sup>

<i>Add</i>
$\Delta S$
$z? : \mathbb{N}$
$x' = x + z?$

We will develop a recursive solution to this by using the recursive procedure synthesis rule given above. So we need to note the following special cases of  $(\lambda z? \bullet Add)[n]$  (by  $\beta$ -equality this is  $Add[z?/n]$ , or  $Add[n]$ , since the parameter is clear from the context).

First we have  $Add[0]$ :

$\Delta S$
$x' = x$

and then  $Add[n + 1]$ :

$\Delta S$
$n : \mathbb{N}$
$x' = (x + n) + 1$

The former is immediately implemented by `skip` (since for any binding  $\sigma$  we have  $\sigma.x = \sigma'.x'$ ).

The latter can be expressed as the composition of two simpler schemas,  $Add[n]$ :

$\Delta S$
$n : \mathbb{N}$
$x' = x + n$

and:

<i>Succ</i>
$\Delta S$
$x' = x + 1$

<sup>5</sup> More extensive and persuasive examples are to be found in [12].

<sup>6</sup> All preconditions in this example are *true*. In such circumstances we omit them from the schema.

That is, we can show that:

$$Add[n] \circ Succ \sqsupseteq Add[n + 1]$$

This utilises the refinement inequation for composition that we introduced at the end of section 3.

The recursive procedure synthesis rule provides  $p[n]$  as an implementation of  $Add[n]$  by assumption. We can implement  $Succ$ , using the assignment rule, by  $x := x + 1$ . Then these, in sequence, implement the composition using the composition rule. And that completes the derivation.

Of course, we can also write the derivation formally as a deduction in the program/refinement logic:

$$\frac{\overline{\text{skip} \in Add[0]} \quad \overline{p[m]; x := x + 1 \in Add[m + 1]}}{p \in Add} \quad \begin{array}{c} \delta_0 \\ \vdots \end{array}$$

where  $\delta_0$  is:

$$\frac{\overline{Add[n] \circ Succ \sqsupseteq Add[n + 1]} \quad \overline{p[m] \in Add[m]} \quad \overline{x := x + 1 \in Succ}}{\overline{p[m]; x := x + 1 \in Add[n] \circ Succ}} \quad \overline{p[m]; x := x + 1 \in Add[m + 1]}$$

and where the procedure  $p$  is thus given as follows:

```

proc p[z?] cases z? in
  0 : skip
  m + 1 : p[m]; x := x + 1
endcases

```

## 7 Conclusions, related and future work

The various proposals we have made should be thought of as establishing a new approach to schema-algebra-based specification: that is, in a *Z style*. In fact, very little of *Z* remains beyond the very *idea* that a useful specification language would address the issue of scaling-up by introducing means by which large specifications can be constructed from smaller components.

We have argued that, in order to integrate specification with design and with implementation, the issue of compositionality, that is, monotonicity, is a central concern. Unfortunately *Z* is not compositional with respect to its standard model of refinement. Related work, to which we have referred earlier, indicates that there is no alternative approach to refinement which would repair this deficiency: a variety of models, built on the standard partial relation semantics, are all equivalent and share the same weaknesses. Any rational reconstruction of *Z*, which requires a monotonic schema logic, must replace the standard semantics (and, as a consequence, the equational logic).

We have also argued that the single-predicate syntax of *Z* schemas is also a problem when considered in the context of design and implementation: *Z* substitutes the notion

of (logical) feasibility for (syntactic) precondition – a much stronger notion. In program development, discharging feasibility conditions is generally more trouble than discharging preconditions, and is rarely beneficial. These observations lead directly to a specification approach in the spirit of the refinement calculus and *B* rather than *Z*.

We have outlined (no more) two possible approaches for compositional schema-based specification: one based on a sets-of-implementations model for schema, and the other based on the lifted-totalisation approach. The former is a well worked out approach (see [12]), the latter, relational approach, is newer and more experimental. Both are fully monotonic and both lead to schema-calculi which are distinct from that of *Z*: the equational logic is replaced by an inequational logic of refinement.

This raises a number of interesting questions which this scene-setting paper has only briefly touched on. First, pragmatically, how does one specify systems in this framework – evidently the *meaning* of conjunction (and so on) is not that of *Z*, and this will lead to distinctive approaches. Second, what schema operators are natural and appropriate in these models? After all, there is no reason to suppose that these would be those of *Z* for at least two reasons: first, because the semantics is distinct and second, because the *purpose* of schema operators is generalised beyond specification to include design and implementation. Indeed, we have described two such operations (abstraction and application) in each model whose purpose is linked with design and implementation rather than specification.

The space of possibilities here is quite large, and the territory relatively unexplored. There remain, then, a number of theoretical and pragmatic questions for the future. This paper has aimed only to review current knowledge through reference to recent research results, and to point out some interesting avenues for future, detailed, investigation.

There are other approaches which consider program development from *Z* specifications, the most ambitious being ZRC [6] and [5]. In this approach *Z* is given a WP-semantics equivalent to its standard semantics and this is used to integrate specification with refinement calculus [16]. The passage from *Z* specifications to specification statements induces preconditions in the standard way (as feasibility conditions) – we have discussed the practicalities of this above. Use of schema operators is hampered (there are quite strong side-conditions on rules involving schema operators) because the refinement calculus solves the *frame-problem* by insisting (it is a trivial consequence of the WP-semantics) that observations outside this frame do not change. This does not sit well with component schemas which have overlapping or disjoint frames. Despite these problems the approach is well-developed and has much to recommend it: the approaches described here (which specifically address the disadvantages just listed, but doubtless have limitations

of their own) can be carefully compared in the future with ZRC.

## 8 Acknowledgements

Moshe Deutsch and Besnik Kajtazi are both supported by ORS awards. The authors would like to acknowledge useful conversations with Steve Dunne, Lindsay Groves and Steve Reeves in connection with the work reported here.

## References

- [1] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.
- [2] J. R. Abrial. *The B-Book*. Cambridge University Press, 1996.
- [3] R. Barden, S. Stepney, and D. Cooper. *Z in Practice*. Prentice Hall, 1994.
- [4] M. Beeson. Proving programs and programming proofs. In *Logic, Methodology and Philosophy of Science*, pages 51–82. Elsevier, 1986.
- [5] A. Cavalcanti. *A Refinement Calculus for Z*. PhD thesis, University of Oxford, 1997.
- [6] A. Cavalcanti and J. C. P. Woodcock. ZRC – a refinement calculus for Z. *Formal Aspects of Computing*, 10(3):267–289, 1998.
- [7] M. Deutsch, M. C. Henson, and S. Reeves. An analysis of total correctness refinement models for partial relation semantics I. *Logic Journal of the IGPL*, 11(3):287–317.
- [8] M. Deutsch, M. C. Henson, and S. Reeves. Operation refinement and monotonicity in the schema calculus. In D. Bert, J. Bowen, S. King, and M. Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 103–126. Springer-Verlag, June 2003.
- [9] L. Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University, 2000.
- [10] L. Groves. Refinement and the Z schema calculus. In *REFINE 2002: Refinement Workshop*. BCS FACS, July 2002.
- [11] J. He, C.A.R Hoare, and J.W. Sanders. Data refinement refined. In G. Goos and J. Hartmanis, editors, *European Symposium on Programming (ESOP '86)*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.
- [12] M. C. Henson and S. Reeves. A logic for schema-based program development. *Formal Aspects of Computing*, 15(1).
- [13] M. C. Henson and S. Reeves. Investigating Z. *Logic and Computation*, 10(1):43–73, 2000.
- [14] S. King. Z and the Refinement Calculus. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM '90 VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 164–188. Springer-Verlag, April 1990.
- [15] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North Holland, 1982.
- [16] C. C. Morgan. *Programming from Specifications*. Prentice Hall International, 2nd edition, 1994.
- [17] S. Stepney, F. Polack, and I. Toyn. Patterns to guide practical refactoring: examples targeting promotion in Z. In D. Bert, J. P. Bowen, S. King, and Marina Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2003.
- [18] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall, 1996.