A Categorical Approach to Structuring and Promoting Z Specifications

Pablo F. Castro[1,3], Nazareno Aguirre[1,3], Carlos López Pombo[2,3], and Tom Maibaum[4]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina. Email: {pcastro, naguirre}@dc.exa.unrc.edu.ar
[2] Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina. Email: clpombo@dc.uba.ar
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
[4] Department of Computing & Software, McMaster University, Hamilton (ON), Canada. Email: tom@maibaum.org

Abstract. In this paper, we study a formalisation of specification structuring mechanisms used in Z. These mechanisms are traditionally understood as syntactic transformations. In contrast, we present a characterisation of Z structuring mechanisms which takes into account the semantic counterpart of their typical syntactic descriptions, based on category theory. Our formal foundation for Z employs well established abstract notions of logical systems. This setting has a degree of abstraction that enables us to understand what is the precise semantic relationship between schemas obtained from a schema operator and the schemas it is applied to, in particular with respect to property preservation.
Our formalisation is a powerful setting for capturing structuring mechanisms, even enabling us to formalise *promotion*. Also, its abstract nature provides the rigour and flexibility needed to characterise extensions of Z and related languages, in particular the heterogeneous ones.

1 Introduction

The intrinsic preciseness of formal specification languages usually lead to very detailed, large descriptions of software systems. Therefore, appropriate mechanisms for structuring specifications are essential in contributing to the scalability of formal specification, and the usefulness of a formal method. This has been acknowledged by formal method developers, and many formal notations, e.g. B, Z and related languages, put a strong emphasis in structuring [1, 25]. In the case of Z, there exist several mechanisms for structuring specifications, called *schema operations*, since they operate on schemas, the basic modularisation units of a Z specification. Traditionally, structuring mechanisms in Z are captured *syntactically*, i.e., their semantics are understood as syntactic transformations over the composed specifications [25]. This approach, although sound, makes it difficult to understand the precise relationship between the composite specifications and their components. Indeed, understanding how the properties of a specification

are involved in properties of another specification including it, is an issue that generally needs to be analysed in an ad-hoc way in every concrete structuring situation, due to the syntactic semantics of the structuring mechanisms employed. This is particularly the case with *promotion*, a Z specification structuring technique. Promotion is typically used in order to compose specifications, and in particular to incorporate multiple instances of a component into a global system state. Since the use of promotion usually involves mapping a "local state" into a "global state" where multiple local states are subsumed, understanding the relationship between the local and global states is particularly difficult.

In this work, we study a formalisation of Z structuring mechanisms, including *promotion*, which in contrast to the syntactic approach to structuring mechanisms semantics, provides strong ties to the semantic counterpart of these mechanisms' syntactic description. This formalisation is based on category theory, and consists of a mathematical foundation for Z and its usual schema operators, making use of institutions and institution representations. This setting has a degree of abstraction that enables us to understand what is the precise semantic relationship between schemas obtained from a schema operator and the schemas it is applied to (in the case of promotion, between basic and promoted schemas), in particular with respect to property preservation. Our formalisation is targeted to Z. A main reason for this is that Z is a mature and widely known formal notation, used in many industrial projects, and supported by analysis tools. Moreover, Z has been used as the basis for other formalisms, such as B, Z++ and Object-Z. In these languages, structuring mechanisms based on or inspired by promotion are also present, and also syntactically captured (in particular, the mechanisms for characterising the notion of class in the object oriented extensions of Z). By basing our formalisation on Z, we make our results also relevant to these other languages.

Our formal foundation for structuring in Z has practical advantages. It leads to explicit semantic relationships between component schemas and the composite schemas they are part of, which can be exploited to *promote* reasoning. Furthermore, if a schema is restricted to a particular "simpler" logic (e.g., a decidable fragment of the Z notation), then one can reason in this simpler setting (perhaps via some automated tool) and then promote the obtained properties to the larger, composite specification in which the schema is involved, and where more expressive constructs may be used. Also, our foundations for promotion require dealing with schemas as types; our semantics of this facility, interpreted as a manipulation of the logical theories that schemas represent, makes it non dependent on higher order logic (as opposed to schema types as treated in [17]), constituting a potential benefit for automated reasoning. Finally, the abstract nature of our characterisation, at a level of abstraction that allows for a view of logical systems as building blocks, provides the rigour and flexibility needed to characterise not only Z but also its related languages and extensions, in particular the heterogeneous ones. It then provides the formal foundations for correctly composing Z with other formalisms, and a setting where one is able to formallly reason about the resulting heterogeneous specifications.

2 A Brief Overview of Z

Z is a formal notation based on mathematical logic and set theory. It is often regarded as being *model based*, since specifications in the language describe systems behaviour via *models*, typically involving data domains and operations on these domains [25]. Such models are expressed in terms of well defined types, including a rich set of built-in types such as the typical numerical domains, sets, sequences, tuples, relations and functions, etc. Z specifications are structured around the notion of *schema* [25]. Essentially, a schema defines a set of typed variables, whose values might be constrained. A schema has a *declaration* section, and a *constraint* (or predicate) section. This extremely simple notion is powerful and convenient for defining data domains and operations on these, as formal models of systems. As a first example, suppose that we need to specify a game similar to Risk, consisting of players whose goal is to conquer territories in a map. For simplicity, let us suppose that territories are labelled by natural numbers, identifying each territory. We might start by defining players, indicating the territories they own. In Z, this is achieved by the following schema:

Player

$$owns : \mathbb{P}\,\mathbb{N}$$

This is a very simple schema, that has an empty predicate part (no special constraints on the variables). Basic operations for a player are settling in a territory, and leaving an occupied territory. In Z, operations are also captured by schemas; schemas characterising the settle and leave operations are the following:

Settle

$$\Delta Player$$
$$t : \mathbb{N}$$
$$t \notin owns$$
$$owns' = owns \cup \{t\}$$

Leave

$$\Delta Player$$
$$t : \mathbb{N}$$
$$t \in owns$$
$$owns' = owns \setminus \{t\}$$

In these schemas, $\Delta Player$ indicates that two copies of the schema *Player* are incorporated into *Settle* and *Leave*, one exact copy of *Player* and the other with its variables renamed by priming. This is done in order to capture the effect of settling (resp. leaving) as a relation between "pre" states of the player (the unprimed variables) and the "post" states of the player, resulting from settling on (resp. leaving from) a territory. Additional variables, in this case representing parameters of the operations, are incorporated and constrained in the predicate part of the schemas. When defining a schema in terms of another one, constraints from the used schema are made part of the constraints of the using schema; for instance, constraints from *Player* and *Player'* (coming from $\Delta Player$) are part of the *Settle* schema (although in this case no actual constraints are incorporated, because the used schemas had no constraints). According to the denotational semantics of Z, a model for a schema is an assignment, that provides values in the corresponding types for the variables in the schema, and satisfies the

predicate part of the schema [21]. That is, a model provides actual values for the variables in a schema. Notice for instance that, for the case of *Player*, all possible models of the schema capture the "state space" for the player.

Z also features schema structuring operations, that is, operations that enable one to define schemas based on other existing schemas. A rather simple one is *schema composition*. Suppose that we would like to define an operation to capture the situation in which a player exchanges one territory for another one, i.e., it leaves a territory and settles in another one. Such an operation can be defined using schemas *Leave* and *Settle*, via a simple composition:

$$Exchange \mathrel{\widehat=} Leave[t_1/t] \fatsemi Settle[t_2/t]$$

This composition (slightly complicated with the renamings necessary for the composition to distinguish the t variables in the two schemas) captures the state change produced by applying the second operation to the state resulting of the application of the first operation.

Promotion is another structuring mechanism of Z. It enables one to *promote* definitions given in terms of "local states", to definitions of a "global state", often composed of various instances of the local state [25]. As an example, suppose that we define the game state, using our previously defined *Player* schema:

Game

$$ps : \mathbb{P}\, Player$$
$$ts : \mathbb{P}\, \mathbb{N}$$

$$ts \neq \emptyset$$
$$\forall\, p : ps \bullet p.owns \subseteq ts$$
$$\forall\, p_1, p_2 : ps \bullet p_1 \neq p_2 \Rightarrow p_1.owns \cap p_2.owns = \emptyset$$

This schema explicitly indicates who are the players of the game (ps), and the territories composing the map (ts); it also constrains the valid states of the game to nonempty sets of territories, and prevents players from sharing the occupation of a territory. We have already defined game related operations *Settle* and *Leave*, but we have done so for *Player*. We would like to be able to *promote* these "local" operations to the "global" state characterised by *Game*, instead of having to redevelop them as operations on *Game*. In order to do so, one needs to define a *promotion schema*, i.e., a schema relating the local and global states:

PromotePlayer

$$\Delta Game$$
$$\Delta Player$$
$$p : Player$$

$$p = \theta Player \wedge p \in ps$$
$$ts = ts'$$
$$ps' = ps \setminus \{p\} \cup \{\theta Player'\}$$

Notice that this schema indicates how a state change of a single player is embedded into a state change for the global state of the game. Now, one can promote the *Settle* operation to the system level, as follows:

$$GameSettle \mathrel{\widehat=} \exists\, \Delta Player \bullet Settle \wedge PromotePlayer$$

The existential quantification in this definition has the purpose of hiding the "local state", which by the restrictions in the *PromotePlayer* schema is already embedded into the state of the game. This makes *GameSettle* an operation exclusively on the state of the game.

3 A Categorical View of Z

Let us recall some basic definitions of category theory. A category is a mathematical structure composed of two collections: the collection of objects: $a, b, c, \ldots$ and the collection of arrows (or morphisms): $f, g, h, \ldots$ between them. An arrow has a domain and a codomain, and we write $f : a \to b$ to indicate that a (resp. b) is the domain (resp. codomain) of f. We have two basic operations involving arrows: the *identity*, that given an object a produces an arrow $id_a : a \to a$, and the *composition*, which given arrows $f : a \to b$ and $g : b \to c$, returns an arrow $f\,;\ g : a \to c$. Identity arrows satisfy: $f\,;\ id_b = f$ and $id_a\,;\ f = f$, for every $f : a \to b$. The composition of arrows is associative. A *functor* is essentially a homomorphism between categories. The most natural example of a category is **Set**, made up of the collection of sets and the collection of functions between sets. We refer the interested reader to [2], for an introduction to category theory. We will assume throughout the paper that the reader has some basic knowledge of category theory.

As we already discussed, a schema defines a set of typed variables, and provides constraints on these variables. Formally, a schema corresponds to a tuple $\langle N, T, \Sigma, \Phi \rangle$ composed of a name N, a set of given types T, a signature Σ (the set of typed variables declared in the schema) and a set Φ of formulas, constraining these variables [22]. For the sake of simplicity, we omit the name and the set T of types when no confusion is possible. The formulas of the predicate part Φ of a schema are higher-order formulas (since Z includes recursive datatypes, lambda expressions, quantification over relations and other elements that go beyond first-order logic's expressiveness) defined over the variables in the declaration part of the schema.

In order to study Z structuring, we need to look at the way schemas relate to each other. A *morphism* between two schema signatures $\tau : \Sigma \to \Sigma'$ is a mapping between symbols that preserves types. Examples of signature morphisms are symbol substitutions (renaming variables in a signature), and embeddings of a signature into another one. Signatures and signature morphisms constitute a category.

Theorem 1. *The structure* $\mathbf{Zign} = \langle S, M \rangle$*, where S is the set of Z signatures and M is the set of signature morphisms, is a category.*

Signature morphisms can be straightforwardly extended to schema morphisms:

Definition 1. *A schema morphism $\tau : \langle \Sigma, \Phi \rangle \to \langle \Sigma', \Phi' \rangle$ is a signature morphism $\sigma : \Sigma \to \Sigma'$ that satisfies the following condition:*

$$\forall\, \phi \in \Phi \bullet \Phi' \vdash \sigma^*(\phi)$$

where σ^* is the inductive extension of σ to formulas, obtained by preserving logical symbols, and $\Phi \vdash \phi$ expresses that ϕ can be proven from Φ using the deductive machinery of Z.

Essentially, a schema morphism is a mapping between logical theories [11]. Using schemas and schema morphisms, a category can be defined:

Theorem 2. *The structure* **Zchem** $= \langle Sch, Tr \rangle$*, where Sch is the set of Z schemas and Tr is the set of schema morphisms, is a category.*

The category **Zchem** enables us to capture the way in which Z schemas relate to each other, and in particular how these are connected in the definition of a structured specification.

In order to clarify the above view of signatures and schemas as objects in a category, consider the diagram in Figure 1. This diagram involves two simple schemas, one of them being our previous *Game* schema, and the other being a simple schema defining a nonempty set of natural numbers. The schema morphism in this diagram shows that the simpler schema is embedded, after translation, into the schema *Game*. Notice that, for this morphism to be correct, one must be able to prove that the translation of $\#ns > 0$ (i.e., $\#ts > 0$) is a consequence of the constraints in the *Game* schema, which is trivial. After this simple example, the reader familiar with Z may notice that schema morphisms subsume the notion of schema strengthening. Models complement the picture of schemas and schema morphisms. An interpretation for a given signature is a valuation of its variables (a function which maps variables to values). For instance, an interpretation for the signature of *Numbers* is simply a nonempty set of natural numbers. Now, given a signature, a model of it is a nonempty collection of interpretations for its variables. In some sense, this enables a *loose* semantics for schemas: each schema denotes a collection of interpretations, in contrast to the more usual *tight* semantics, where a schema denotes only one interpretation. This semantics will be in particular useful for formalising promotion (see section 4). An example of a model for *Numbers* is shown below it in Fig. 1, using a notation borrowed from [25]. This model maps ns to the sets $\{0, 1\}$ and $\{2, 3\}$. Given a schema morphism $\tau : S_1 \to S_2$, this morphism induces a mapping $()_{|\tau} : Mod(\Sigma') \to Mod(\Sigma)$ between models of S_2 and models of S_1 [15]. This mapping builds *reducts* [10], i.e., given a model of the "larger" schema, it removes from the model all the parts that are unnecessary to interpret symbols originating in S_1, obtaining a model of the smaller schema. An example of a reduct, obtained from a model of the schema *Game*, is also shown in Figure 1. Given an interpretation I, we say $I \vDash \phi$ if I satisfies the property ϕ; and given a model M and a collection of formulas Φ, we say $M \vDash \Phi$, if for every $I \in M$ and $\phi \in \Phi$, we have $I \vDash \phi$. Models of schemas are those satisfying the predicate part of the schema. As usual, we will use the notation $M \vDash \Sigma$ (resp. $M \vDash S$) to express that M is a model of a signature Σ (resp. of a schema S). It is worth remarking the following property, which relates signature morphisms with models and formulas:

$$M_{|\sigma} \vDash \phi \Leftrightarrow M \vDash \sigma^*(\phi),$$

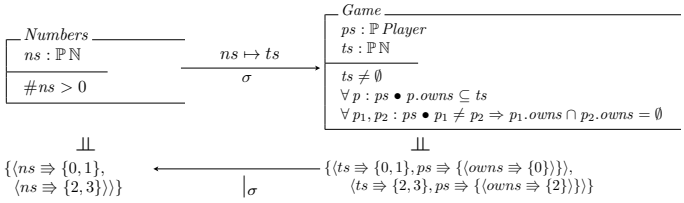


Fig. 1. An example involving schemas, schema models, a schema morphism and the corresponding model reduct.

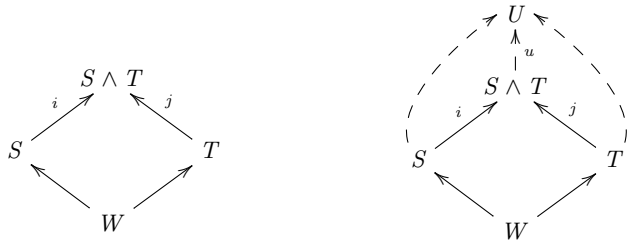


Fig. 2. Schema conjunction as a pushout.

where $\sigma : \Sigma_1 \to \Sigma_2$ is a signature morphism, M is a model of Σ_2 and $M_{|\sigma}$ is the reduct of M to the syntax of Σ_1. This property expresses that syntactic changes of formulas via signature morphisms do not affect the notion of truth. This is a main characteristic of an *Institution* [15]. Indeed, regarding Z, we have the following theorem.

Theorem 3. *The structure* **Z** *composed of:* (i) *the category* **Zign**, (ii) *the functor sen* : **Zign** $\to$ **Sen**, *that sends each signature to its set of formulas,* (iii) *the functor Mod* : $\mathbf{Zign}^{op} \to$ **Cat**, *that sends each signature to the category of its models*[5], *and* (iv) *the collection of relations* $\models_\Sigma$ *(satisfaction relations relating models of a signature to formulas of the signature), is an* Institution.

Let us continue with our categorical characterisation of Z concepts. The main mechanism for putting two Z schemas together is *schema conjunction*. In a categorical setting, the corresponding way of combining two schemas is captured by a categorical operation called *pushout*. The diagram in Figure 2 depicts what a pushout is, and how it captures schema conjunction. In this diagram, W is the common part of S and T, i and j are identity arrows, and $S \wedge T$ is obtained by putting S and T together, keeping only once the common part (exactly what schema conjunction does [25]). The pushout is *minimal*, in the sense that for any other schema U such that we have arrows from S, T to it, we can obtain a unique arrow from $S \wedge T$ to U such that the diagram shown in Figure 2 commutes.

[5] $\mathbf{Zign}^{op}$ denotes the dual category of **Zign**, obtained by reversing arrows. This is needed since reducts and morphisms go in different directions.

Another useful operation over schemas is symbol renaming, in particular renaming by *priming*. Categorically, this schema operation corresponds to an *endofunctor* $(-)' : \mathbf{Zchem} \to \mathbf{Zchem}$, the straightforward extension to schemas of the endofunctor $(-)' : \mathbf{Zign} \to \mathbf{Zign}$ which maps every symbol in a signature to its primed version.

As we explained in the previous section, in a Z specification one usually defines operations via particular schemas, relating other schemas describing domains. In our categorical view of Z, operations correspond to a particular class of diagrams, of the form shown in Figure 3 (a), where A and B are the related "domain" schemas, and C is the operation schema. Such a diagram is indeed a categorical diagram in the category **Zchem**, called a *cospan*. In particular, an operation for a system S (captured as a schema) is typically specified as a schema over ΔS, i.e., over the conjunction of S and S', where S' represents the "post" state of S, i.e., the state after the operation has been executed. Such an operation is also a cospan, and has the form shown in Figure 3 (b).

Let us more precisely formalise the concept of operations.

Definition 2. *An operation is a cospan in* **Zchem** *of the following form:*

$$S \to Op \leftarrow S'$$

We use the notation $Op : S \Rightarrow S'$ *to express the above diagram.*

Operations modifying the state S of a system (captured as a schema) are usually defined over ΔS. ΔS can also be captured categorically:

Definition 3. *Given a schema* S*, we denote by* ΔS *the coproduct of* S *and* S'*, where* S' *is the result of applying the priming functor to schema* S*.*

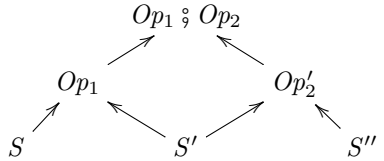
The coproduct is a pushout of two schemas S_1 and S_2 with no common part (i.e., in the figure above we set $W = \langle \emptyset, \emptyset \rangle$); that is, for any other schema combining S_1 and S_2 (meaning that we have schema morphisms from S_1 and S_2 to the combined schema), there exists a unique schema morphism u from the coproduct to this combined schema that makes the diagram involving these schemas and the schema morphisms corresponding to the combinations commute. This situation is described in Figure 4, for the case of ΔS, the coproduct of S and S'.

We have used an arrow notation for cospans, in our characterisation of Z operations. In fact, cospans can be thought of as arrows (or morphisms), which are composed by applying pushouts [4]. This is the way schema composition is categorically captured.

Fig. 3. Cospans, and Z operations as cospans.

Fig. 4. Categorical definition of ΔS as a coproduct.

Definition 4. *Given two operations $Op_1 : S \Rightarrow S'$ and $Op_2 : S \Rightarrow S'$ we define the operation $Op_1 \fatsemi Op_2$ as follows:*



where the tip of the diagram is obtained by means of pushouts, and Op_2' is built up by applying the functor $(-)'$ to Op_2.

Another useful construction in Z is the ΞS operation. This operator on schemas denotes a *skip* operation. That is, it is a special case of ΔS, in which S and S' are identical. This schema operator can also be defined (up to isomorphism) in a categorical way.

Definition 5. *$\Xi S : S \Rightarrow S'$ is a schema that satisfies: $\Xi S \fatsemi Op \cong Op \fatsemi \Xi S \cong Op$, for every operation Op, where $S \cong S'$ expresses that there is an isomorphism between the corresponding schemas.*

Given schemas S and S', we have a category $OP(S, S')$ where the objects are the operations between S and S' and the morphisms are the schema morphisms between the corresponding cospans. This construction is called a *bicategory* [4]. An important point is that we can think of our category of schemas as having two different kinds of arrows, one representing schema morphisms (schema embeddings after translation), and another one capturing Z operations (as cospans), with $\fatsemi$ working as the composition for the latter.

Definition 6. ***Zpec*** *is the* bicategory of Z specifications*, defined as the structure composed of:*

- *The set of schemas as its set of objects.*
- *For each pair of schemas S, S', the category $OP(S, S')$ of cospans between S and S' (called 1-cells), and morphisms between cospans (called 2-cells).*
- *The composition between 2-cells is defined as usual by using the composition (i.e., pushouts) of cospans (denoted by $\fatsemi$).*

Summarising, a Z specification is a collection of schemas $S_0, \ldots, S_n$ together with a set of cospans $Op_i : S_i \Rightarrow S_i'$, all "living" in the bicategory of Z specifications. An example illustrating schemas and operations, and their relationships, is shown in Figure 5, as a diagram in **Zpec**.

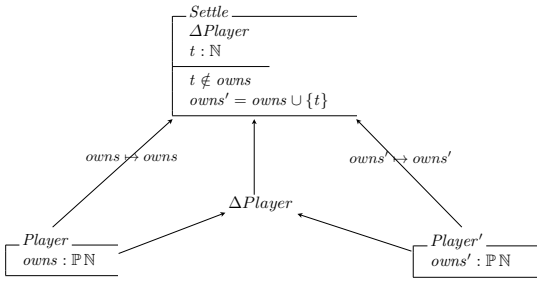


Fig. 5. A Z specification as a categorical diagram in **Zpec**.

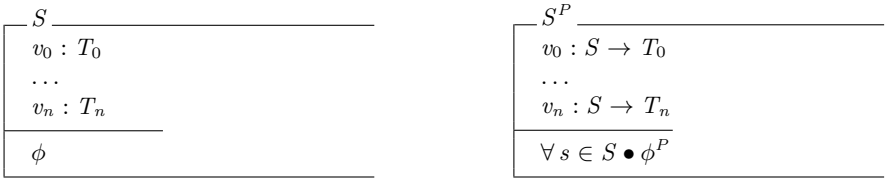


Fig. 6. A schema and its manager construction.

4 Schemas as Types and Promotion

Let us now concentrate on analysing *promotion*. A key feature of Z, that facilitates promotion, is the use of schemas as types. In order to do this, we need to spice up our categorical framework with some additional machinery. Basically, we introduce the concept of schema manager, which conveys the idea of schema instances. Essentially, a manager of a component C is a component that intuitively provides the behaviour of various instances of C, and usually enables the manipulation of these instances. We will deal with the possibility of interpreting schemas as types in a way that differs from the established mechanism to do so, presented in [25]. Our approach consists of building a manager specification. Consider the schemas in Figure 6; the one on the left represents an arbitrary schema, involving $v_0 : T_0, \ldots, v_n : T_n$ as its typed variables. The schema on the right represents the *manager* for the previous schema, where ϕ^P is obtained from ϕ by adding the parameter s of type S to each variable. For the schema on the right, S does not represent the schema on the left; instead, it is simply a fresh *given type*, although for simplicity we maintain for this given type the same name as for the schema it represents.

An example of the use of managers is shown is Figure 10. In this figure, $Numbers^P$, the manager of *Numbers*, is used to provide semantics to the use of schema *Numbers* as a type (to be explained later on).

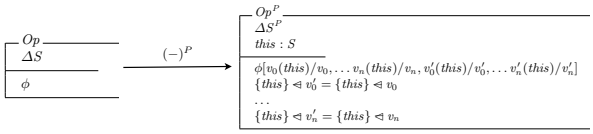


Fig. 7. Operation promotion using managers.

We can define a similar transformation over operations. Consider the schemas in Figure 7; the schema on the left is the definition of an operation, where $v_0, \ldots, v_n, v'_0, \ldots, v'_n$ are the variables of S and S', respectively. We introduce the schema on the right; that is, we add a parameter, in this case named *this*, representing the instance to which the operation is applied. This situation is graphically depicted as a categorical diagram in Figure 8. Therein, the dashed arrows denote the application of the transformation described above. The translation $(-)^P$: **Zpec** $\rightarrow$ **Zpec** is a functor, which maps schemas to promoted schemas, and operations to promoted operations. We can define it in three parts:

- A functor $(-)^P$: **Zign** $\rightarrow$ **Zign**, which translates signatures in the way described above.
- A functor $(-)^P : OP(S, S') \rightarrow OP(S^P, (S^P)')$, that translates operations to promoted operations. (For the sake of simplicity we use $(-)^P$ for naming these two functors.)
- The canonical extension of $(-)^P$ to formulas, as explained above.

The following theorem can be proven by resorting to the definition of $(-)^P$.

Theorem 4. $(-)^P$: **Zpec** $\rightarrow$ **Zpec** *is a lax functor.*

Lax functors are morphisms between bicategories; this means that promotion is coherent with respect to identities and composition of operations. Moreover, given a model M of Σ^P we can define a corresponding model M_D (a degraded model), which forgets the new sort introduced. In Figure 9, a simple example of a mapping between schemas and their models is shown, to illustrate these ideas. These kinds of mappings are called *institution representations* [23], and are mappings between logical systems. Intuitively, a collection of schemas and the relations between them conform a logical system. An institution representation

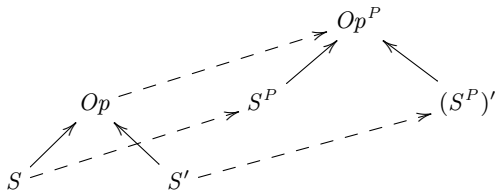


Fig. 8. Categorical diagram depicting operation promotion.

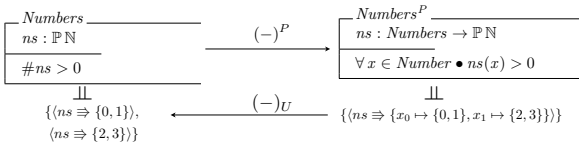


Fig. 9. Example of mappings between schemas and models

allows us to move inside the same system but adding certain useful features, while keeping the basic properties of these schemas.

The operation of using a schema as a type can be understood as a kind of schema inclusion. Consider for instance the schema given in Figure 10 (a); it defines the end state of a game where the player needs to conquer territories 0 to 6. Notice that the actual semantics of this schema can be defined using the schema manager $Numbers^P$ introduced above, simply by including $Numbers^P$ in the schema. This has a self evident categorical interpretation, and the existence of an arrow between *Numbers* and *EndGame* relates them both syntactically and semantically. This resulting diagram is shown in Figure 10 (b), where $result.ns$ is just syntactic sugar for $ns(result)$.

This simple approach based on managers allows us to deal with schemas as types. We just dealt with "single instances", but the approach is also suitable for dealing with indexed instances of a schema, as is usual when using promotion. For instance, consider a game where we have various players, each player with its own set of territories. A schema illustrating this situation, with a promoted operation and showing the role of managers, is shown in Figure 11.

4.1 Promotion as an Institution Representation

Institution representations were introduced informally above, where we argued about their need for capturing promotion. As institutions are an abstract characterisation of logical systems, institution representations capture the notion of embedding of a logical system into another one [23]. The logical machinery of Z used for describing states and operations constitutes an institution, and the op-

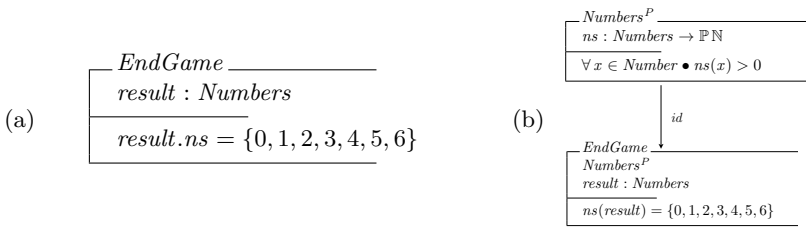


Fig. 10. Using managers as types

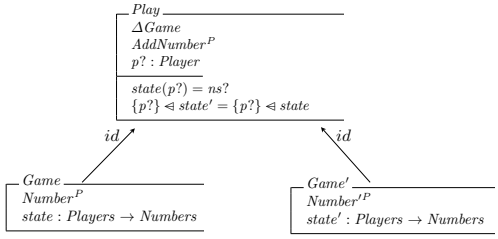


Fig. 11. Using managers for promoting an operation.

eration of promoting schemas corresponds to an institution representation from this institution to itself. The key elements involved in the promotion process are:

- The definition of a mapping (functor) $(-)^P$: **Zign** $\to$ **Zign**, mapping a signature to its promoted signature.
- The definition of a mapping (natural transformation) $(-)_D : Mod\,;(-)^P \to Mod$, mapping models of promoted signatures to models of the original signature.
- The definition of a mapping (natural transformation) $(-)^P$: **Sen** $\to$ **Sen**$\,;(-)^P$ mapping formulas of the original signature to formulas of the promoted signature.

These mappings satisfy the property $M \models \phi^P \Leftrightarrow M_D \models \phi$. That is, a model of a promoted signature satisfies a promoted property if and only if the degraded model satisfies the original property. A graphical representation of this situation is shown in Figure 12. To clarify this diagram, suppose that we have a translation from one schema signature to another schema signature (named σ). Notice that reducts move in the opposite direction of translations (this explains the $(-)^{op}$ in the definition of institutions). Then, if we take a reduct of a promoted schema, and so we take the degraded model (the right path of the diagram), we obtain the same model as if we take the degraded model first and then take the reduct (the left path in the diagram). This ensures the coherence between the operations of strengthening and promotion in Z, which is guaranteed by the following theorem.

Theorem 5. *$(-)^P$ and $(-)_D$ are institution representations.*

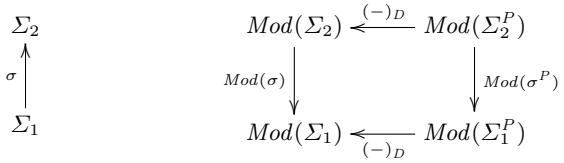


Fig. 12. Institution representations.

5 Heterogeneous Z Specifications and Structuring

Following the recent trend in Software Engineering that favours a "multiple views" approach to specification and design, the Z notation has been extended in various ways, in combination with other notations. Some of these extensions are Z-CSP [14] (Z plus the process algebra CSP), and Z plus statecharts [24]. These *heterogeneous* specifications pose new challenges, e.g., for defining appropriate formal semantics for the composite languages, and for providing effective mechanisms to reason about their specifications.

A consequence of the abstract nature of our formalisation of Z, and its structuring mechanisms, is that we can deal with these extensions in a systematic way. Basically, individual formalisms for specifying software systems can be viewed as institutions; indeed, first-order logics [15], temporal logics [15], modal logics [15], Unity-like languages [13] and process algebras [19], all constitute institutions. Our formalisation of Z in an institutional setting, and the wide toolset available from the theory of Institutions, enables us to flexibly combine Z with other formalisms, obtaining extensions of Z with appropriate, well structured semantics[6]. In order to illustrate this nice characteristic of our formalisation, we briefly describe in this section the combination of Z with CSP (structured CSP, as introduced in [19]). The obtained combination is, in essence, similar to the framework Z-CSP, with a well defined structured semantics, that makes the semantic relationships between different (heterogeneous) components of a specification explicit. We make use of the **CSP** (structured CSP) institution. The interested reader can find the details of this formalism in [19]. Signatures in this institution are pairs $\langle A, P\rangle$, where A is an alphabet (used for the communication of processes), and P is a collection of process names. Elements of both A and P have an associated list of typed parameters. A morphism $\langle f, g\rangle : \langle A, P\rangle \to \langle A', P'\rangle$ between two **CSP** signatures consists of an injective function $f : A \to A'$, mapping members of A to members of A' preserving parameters and their types[7], and a function $g : P \to P'$, mapping process names to process names, preserving parameters and their types. The category of **CSP** signatures is called **CSPSig** [19]. A **CSP** theory is a tuple $\langle \Sigma, \pi\rangle$, where Σ is a **CSP** signature, and π is a set of processes in the CSP notation. A model of a theory is given by a set of traces corresponding to the processes of the theory. For the sake of simplicity, we employ a finite trace semantics (as introduced in [19]), although also the failure-divergence semantics is supported in this institution. We have a morphism between models $M_1 \to M_2$ iff $M_2 \sqsubseteq M_1$ (i.e., M_2 is a refinement of M_1). A simple example of a vending machine is described as a **CSP** theory in Fig. 13. Neither communication letters nor processes have parameters in this example. A model of the theory accompanies the example as well.

A new institution **CZP** can be defined using the institutions **CSP** and **Z**. Essentially, we want specifications to have a data part, given in Z with its cor-

[6] The combination of institutions is well studied; see for instance [18].

[7] The use of injective mappings introduces some subtle technical problems when combining specifications. A way of avoiding these problems is described in [19].

$$\begin{array}{l} \{\langle\rangle, \langle coin\rangle, \\ \langle coin, choc\rangle, \\ \langle coin, choc, coin\rangle \\ \ldots\} \end{array} \models \begin{array}{l} A = \{coin, choc\} \\ P = \{VM\} \\ \pi = \{VM = coin \to choc \to VM\} \end{array}$$

Fig. 13. A theory in Structured CSP, and a model of it.

responding operations, and a process part, with each atomic process being associated with an operation as described in the Z part of the specification.

The category **SignCZP** of **CZP** signatures is composed of: *(i)* tuples $\Sigma = \langle \Sigma_{CSP}, \Sigma_Z \rangle$ as signatures, where Σ_{CSP} and Σ_Z are CSP and Z signatures, respectively; *(ii)* a morphism $\sigma : \Sigma \to \Sigma'$ is a tuple of morphisms $\langle f : \Sigma_{CSP} \to \Sigma'_{CSP}, g : \Sigma_Z \to \Sigma'_Z \rangle$. The functor sen_{CZP} is defined as follows:

$$sen_{CZP}(\langle \Sigma_{CSP}, \Sigma_Z \rangle) = \langle sen_{CSP}(\Sigma_{CSP}), sen_Z(\Sigma_Z) \rangle.$$

The functor Mod_{CZP} is defined as follows: *(i)* Given $\Sigma = \langle \Sigma_{CSP}, \Sigma_Z \rangle$, we define:

$$Mod(\Sigma) = \{\langle a_1, \ldots, a_n \rangle, s\rangle \mid \exists\, M \in Mod(\Sigma_{CSP}) : \langle a_1, \ldots, a_n \rangle \in M \wedge s \in Mod(\Sigma_{Op})\},$$

where Σ_{Op} is the signature of the operation $event(a_1) \fatsemi \cdots \fatsemi event(a_n)$. That is, models are execution traces, together with models of the corresponding operation. *(ii)* Given a morphism $\sigma : \langle A, N \rangle \to \langle A', N' \rangle$, the morphism $Mod(\sigma)$ is defined pointwise, using reducts of traces as defined in [19], and reducts of schema valuations as defined in Section 3.

The relation $\models_{CZP}$ is also defined resorting to $\models_{CSP}$ and $\models_Z$ as follows:

$$M \models \langle \pi, \phi \rangle \text{ iff } \pi_1(M) \models \pi \text{ and for every } s \in \pi_2(M) \text{ we have } s \models \varphi.$$

A theory in **CZP** is a tuple $\langle \Sigma_{CSP}, \Sigma_Z, S, Ops, events, \pi \rangle$, where *(i)* $\Sigma_{CSP} = \langle A, N \rangle$ is a signature in **CSP**, *(ii)* Σ_Z is a signature in **Z**, *(iii)* S is a schema $\langle S, \Phi \rangle$, *(iv)* $Ops = \{op_0 : S \Rightarrow S', \ldots, op_n : S \Rightarrow S'\}$ is a collection of operations over the state S, *(v)* $event : A \to Ops$ is a function mapping events to operations, and *(vi)* π is a set of CSP processes. Morphisms between **CZP** theories are straightforwardly defined pointwise.

The relation $\models$ is extended to theories: $M \models \langle \Sigma_{CSP}, \Sigma_Z, S, Ops, event, \pi \rangle$ iff for every $\langle \langle a_1, \ldots, a_n \rangle, \langle s_i, \ldots, s_n \rangle \rangle \in M$ we have that $\pi_1(M) \models \pi$, and $\pi_2(M) \models event(a_1) \fatsemi \cdots \fatsemi event(a_n)$. Figure 15 shows an example of a **CZP** theory.

Promotion can be easily extended to this new institution. We define functor $(-)^P$: **CZPSign** $\to$ **CZPSign**, mapping signatures to signatures, as follows. Given a signature $\langle \Sigma_{CSP}, \Sigma_Z \rangle$, Σ_Z is translated to Σ_Z^P, and Σ_{CSP} is mapped to the following **CSP** signature:

- If $a \in A$, then $a^P = a?(x : S)$, being S the new type introduced in Σ_Z^P,
- If $n \in N$, then $n(x_1 : T_1, \ldots, x_n : T_n)^P = n(x : S, x_1 : T_1, \ldots, x_n : T_n)$.

This functor is extended to sentences in **CZP**: the translation of a process is defined inductively as in Fig. 14, and the translation of Z formulas is defined as in Section 3. Furthermore, we define the mapping $(-)_D$ between models as in Fig. 14. This extension of promotion is also an institution representation:

$(skip)^P \stackrel{\text{def}}{=} skip$
$(stop)^P \stackrel{\text{def}}{=} stop$
$(a \to Proc)^P \stackrel{\text{def}}{=} a?x : X \to Proc^P$
$(?y{:}T \to Proc)^P \stackrel{\text{def}}{=} ?x{:}X?y{:}T \to Proc^P$
$(S \Box Q)^P \stackrel{\text{def}}{=} S^P \Box S^P$
$(S \sqcap Q)^P \stackrel{\text{def}}{=} S^P \sqcap Q^P$
$(S \parallel Q)^P \stackrel{\text{def}}{=} S^P \parallel Q^P$
$(P \mid\mid\mid Q)^P \stackrel{\text{def}}{=} S^P \mid\mid\mid S^P$

$M_D \stackrel{\text{def}}{=} \bigcup_{x \in S} \{\sigma_x \mid \sigma \in M\}$

where σ_x is obtained by deleting the events in the trace where x is not present, similarly for the corresponding interpretation of schemas.

Fig. 14. Promoting basic CSP operators, and degrading traces.

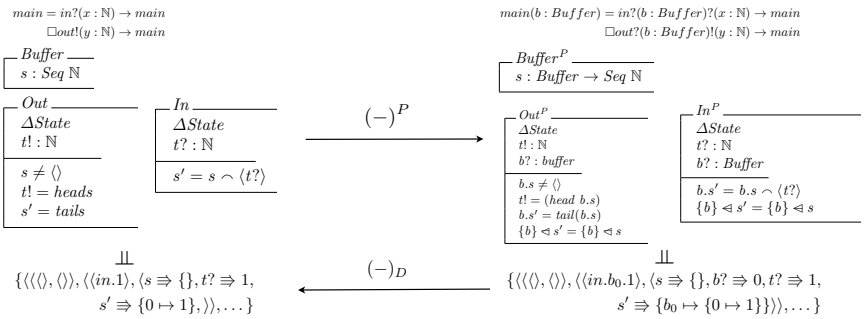


Fig. 15. Promoting CZP specifications.

Theorem 6. *Mappings $(-)^P$ and $(-)_D$ are institution representations.*

Figure 15 shows, using a simple example, how promotion works in this new setting. In this case, we have a standard specification of a buffer with its corresponding process specification. The schemas and the CSP process on the left are promoted to the the corresponding on the right. Via promotion, we obtain a specification with various buffers whose executions interleave.

6 Related Work and Conclusions

We proposed a mathematical foundation for Z and its structuring mechanisms, which makes use of well established abstract notions of logical systems. Indeed, the notions that we used in this formalisation have been employed to structure concurrent system specification languages, algebraic specification languages, and other formalisms [13, 12]. Several alternative approaches to provide a formal semantics to Z can be found in the literature. One of these is the one presented in [21], where schemas are interpreted as axiomatic theories (signatures plus predicates), and the semantics of these axiomatic theories is given by means of varieties; in that work, no semantics is proposed for promotion and the use of schemas as types. In [3], institutions are used for providing semantics to Z specifications; in that work, schemas are captured as logical sentences in an institution,

and therefore a Z specification is viewed as an unstructured set of expressions. In contrast, our approach makes use of theories and morphisms between them in formalising Z designs, thus leading to a well structured categorical semantics of designs. In [7], category theory is used in the definition of a relational semantic framework to interpret Z as well as other specification languages. As in our case, the approach allows for heterogeneous specification; however, the work uses Z simply as an example of a language based on the "state & operations" viewpoint, but it does not show how to deal with Z's structuring mechanisms. In [6], the authors propose a set of rules to manipulate Z schemas; as opposed to our work, these rules are motivated as a means for refactoring specifications. In [16], the authors propose to interpret schemas as types; they build a logical machinery in order to deal with these types. These ideas were adopted in the international ISO standard of Z [17]. Some issues are, in our opinion, not dealt with adequately in that approach; for instance, schema priming is difficult to explain in this context, since a schema and its primed version correspond to different unrelated types. We believe that our approach fits better with the original motivations for Z's schema operators, where priming denotes a purely syntactical operation, an operation also extensively used in other logics for program specification (e.g., in TLA). The interpretation of priming (and related operators) as categorical operations over logical theories provides a simple understanding of Z constructions, with a good separation of concerns between the interpretation of schemas and schema operators, dealing even with promotion, a sophisticated, and widely used, specification structuring mechanism. Moreover, our approach maintains the structure of specifications when providing semantics to them, leading to explicit semantic relationships between component schemas and the composite schemas they are part of, which can be exploited to *promote* reasoning, and with potential benefits for automated reasoning. Finally, our formalisation is at a level of abstraction that allows for a view of logical systems as building blocks. This provides the rigour and flexibility needed to characterise not only Z but also its related languages and extensions, in particular the heterogeneous ones. We have illustrated this point via a formal, well structured, combination of Z with CSP, resulting in a formalism in essence equivalent to the Z-CSP formal method, and "inheriting" the structuring of the composed languages, in particular promotion.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grants PICT PAE 2007 No. 2772, PICT 2010 No. 1690 and PICT 2010 No. 2611, and by the MEALS project (EU FP7 programme, grant agreement No. 295261).

References

1. J.-R. Abrial, *The B-Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.

2. M. Barr and C. Wells, *Category Theory for Computer Science*, Centre de Recherches Mathématiques, Université de Montréal, 1999.
3. H. Baumeister, *Relating Abstract Datatypes and Z-Schemata*, in Proc. of WADT 1999, LNCS 1827, Springer, 1999.
4. J. Bérnabou, *Introduction to bicategories*, in Complementary Definitions of Programming Language Semantics, LNM 42, Springer, 1967.
5. F. Borceux *Handbook of Categorical Algebra: Volume 1: Basic Category Theory*, Enc. of Mathematics and its Applications, Cambridge University Press, 1994.
6. S. M. Brien and A. P. Martin, *A Calculus for Schemas in Z*, Journal of Symbolic Computation, 30(1), Elsevier, 2000.
7. M. C. Bujorianu, *Integration of Specification Languages Using Viewpoints.* In Proc. of IFM 2004, LNCS 2999, Springer, 2004.
8. R. Burstall and J. Goguen, *Putting Theories together to make Specifications*, in Proc. of Intl. Joint Conference on Artificial Intelligence, 1977.
9. P. F. Castro, N. Aguirre, C. G. Lopez Pombo and T. S. E. Maibaum: *Towards Managing Dynamic Reconfiguration of Software Systems in a Categorical Setting*, in Proc. of ICTAC 2010, LNCS 6255. Springer, 2010.
10. C. C. Chang and H. J. Keisler, *Model Theory*, 3rd. Ed., North Holland, 1990.
11. H. Enderton, *A Mathematical Introduction to Logic*, 2nd. Ed., Academic Press, 2001.
12. J. Fiadeiro, *Categories for Software Engineering*, Springer, 2004.
13. J. Fiadeiro and T. Maibaum, *Temporal Theories as Modularisation Units for Concurrent System Specification*, Formal Aspects of Computing, 4(3), Springer, 1992.
14. C. Fischer, *Combining CSP and Z*, Technical Report, University of Oldenburg, 1997.
15. J. Goguen and R. Burstall, *Institutions: Abstract Model Theory for Specification and Programming.* Journal of the ACM, 39(1), ACM Press, 1992.
16. M. Henson and S. Reeves, *Revising Z: Part I - Logic and Semantics*, Formal Aspects of Computing, 11(4), Springer, 1999.
17. J. Nicholls, *Z Notation: Version 1.2*, Z Standards Panel, 1995.
18. T. Mossakowski and A. Tarlecki and W. Pawlowski, *Combining and Representing Logical Systems*, in Proc. of Category Theory and Computer Science 1997, LNCS 1290, Springer, 1997.
19. T. Mossakowski and M. Roggenbach, *Structured CSP - A Process Algebra as an Institution,* in Proc. of WADT 2006, LNCS 4409, Springer, 2006
20. G. Smith, *The Object Z Specification Language*, Advances in Formal Methods Series, Kluwer Academic Publishers, 2000.
21. J. M. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge Tracts in Theoretical Computer Science, 1988
22. J. M. Spivey, *The Z Notation: A Reference Manual.* Prentice Hall, 1992.
23. A. Tarlecki, *Moving Between Logical Systems*, in Proc. of ADT/COMPASS 1995, LNCS 1130, Springer, 1995.
24. M. Webber, *Combining Statecharts and Z for the Design of Safety-Critical Control Systems*, in Proc. of FME 96, LNCS 1051, Springer, 1996.
25. J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*, Prentice Hall, 1996.