A Structure Preserving Encoding of Z in Isabelle/HOL[1]

Kolyang[*], **T. Santen**[‡], **B. Wolff**[*]

Revised Version 1.1

[*]Universität Bremen , FB3
P.O. Box 330440
D-28334 Bremen
{bu,kol}@informatik.uni-bremen.de

[‡]GMD FIRST Berlin
Rudower Chaussee 5
D-12489 Berlin
santen@first.gmd.de

Abstract. We present a semantic representation of the core concepts of the specification language Z in higher-order logic. Although it is a "shallow embedding" like the one presented by Bowen and Gordon, our representation preserves the structure of a Z specification and avoids expanding Z schemas. The representation is implemented in the higher-order logic instance of the generic theorem prover Isabelle. Its parser can convert the concrete syntax of Z schemas into their semantic representation and thus spare users from having to deal with the representation explicitly. Our representation essentially conforms with the latest draft of the Z standard and may give both a clearer understanding of Z schemas and inspire the development of proof calculi for Z.

1 Introduction

Implementations of proof support for Z [Spi 92, Nic 95] can roughly be divided into two categories. In *direct implementations*, the rules of the logic are directly represented by functions of the prover's implementation language. Since hundreds of rules are needed to reason about the mathematical toolkit of Z which defines relations, functions, data types, etc., these implementations are error-prone and tend to be difficult to modify. Moreover, they often lack implementations of advanced deduction techniques like, e.g., higher-order rewriting or resolution.

In contrast to direct implementation, one can choose to *semantically embed* Z in a logical framework. An implementation within a "tactical theorem prover" in the tradition of LCF like *HOL* [GM 93] or *Isabelle/HOL* [Pau 94] is particularly attractive: large libraries, e.g. for set theory, that are proven consistent with the kernel logic can be used to implement and enhance the mathematical toolkit of Z. The machinery of the Isabelle prover, in particular, supports deriving new rules and allows the systematic development of proof calculi for Z. Coming with an open system design going back to Milner, these systems allow for safe user-programmed extensions to support tedious proof tasks often arising in explicit (predicative) type checking or in transformational program development over specifications [KSW 96].

1 This work has been supported by the BMBF projects **UniForM** and **ESPRESS**.

1.1 The Challenge

A lot of criticism from other scientific communities on Z is related to the fact that Z defines itself as a *notation* on the basis of set theory. The essential vehicle to structure Z specifications at the level of concrete syntax is called a *schema*. Schemas have significantly more syntactic flavour than, for example, the notion of a "parameterised abstract data type" used in the algebraic specification community [EM 85].

A *purely* syntactical understanding of structuring would in fact neither be satisfying for theoretical nor for practical purposes. It would mean that structured specifications would have to be *expanded* (flattened) before any semantic treatment. As a consequence, the schema calculus and reasoning at the structural level would be impossible. The resulting formulas would get so large that the advantage of structuring specifications would disappear at the moment where reduction of complexity is of vital importance— namely when reasoning about specifications.

In our approach, schemas are represented as *boolean valued functions*. On the syntactic side, a schema declaration introduces an identifier of a particular class of types together with a particular signature that is used for parsing and pretty-printing purposes. On the semantic side, a schema declaration is a constant definition and may be unfolded during proofs at will. Since schemas are represented as first-class objects of the object logic, the schema calculus of Z can be represented and structured reasoning over specifications is possible.

The major contribution of this paper is to clarify the distinction between syntactic and semantic facets of schemas, at least from a HOL perspective. The practical consequences lie in the implementation of parsing- and pretty-printing machinery as well as in proof support at the level of schemas.

1.2 A Z-Journey Through Related Approaches

Since Z represents a particularly attractive goal for an encoding, several attempts to represent Z in logical frameworks and in particular in higher-order logic have been undertaken. Before we discuss some of them, let us briefly introduce the concepts of Z, crucial for such an embedding. We base the paper on the draft standard [Nic 95], which we will call "The Z Notation" (TZN) hereafter. An alternative would have been to refer to "The Z Reference Manual" (ZRM) [Spi 92], but we felt it more appropriate to follow the proposal for a future ISO standard for Z, although it may still evolve.

Central Concepts of Z. The semantics of Z given in TZN is based on Zermelo-Fränkel set theory (ZF) — which is untyped — but nevertheless Z is a strongly typed language. Each "given set" of a Z specification is associated with a primitive type. Type constructors for power-set types, product types, and schema types correspond to the respective set theoretic constructions. Type correctness with respect to these types is checked by Z type checkers like *f*uzz [Spi 92a]. In the following, we use the term "type" for Z types while "HOL-type" shall refer to types of higher-order logic. Types are not explicitly denoted in Z specifications. Instead, a declaration $x : S$ is a membership statement interpreted as $x \in S$ that implicitly declares x to have the — unique — type of the members of S.

The most important structuring concept of Z are schemas, like A, B and C below:

```
── A ____________
│   x1 :  S1
│   . . .
│   xn :  Sn
├──────────────
│   P1
│   . . .
│   Pm
└──────────────────
```

```
── B ____________          ── C __________
│  A                       │   y : A
│  x1 :  T                 │   c : T
├──────────────            ├──────────────
│  true                    │   Q(y.x1) (y.x2) c
└──────────────────        └──────────────────
```

A schema consists of a list of declarations and a list of — semantically conjoined — predicates which are separated by a horizontal line in the concrete syntax.

A schema may be referenced by its name in subsequent schemas and schema-expressions. This may be *as import* like *A* in *B* (the declaration parts semantically are *sets of declarations*, hence their order and multiple occurrences are irrelevant — but S_1 and *T* must have the same type) or *as a set* like in *C*. Schemas can also serve as predicates like in $B \Rightarrow x_1 \in S_1 \cap T$. Moreover, it is possible to form new schemas in expressions of the schema calculus like $A \Rightarrow B$ or $\exists C \bullet B$ where $\Rightarrow$ is a schema connective and $\exists$ is a schema quantifier. It is important to note that the signature of a referenced schema is unified with the environment on the basis of *lexical* identity: in *B*, the explicitly declared x_1 is identified with the corresponding declaration in *A*.

Approaches to Mechanising Z. Bowen and Gordon's encoding "Z in HOL" [BG 94] represents *A* in the syntax of HOL88 by:

$$x_1 \text{ IN } S_1 \text{ /\\ } \dots \text{ /\\ } x_n \text{ IN } S_n \text{ /\\ } P_1 \text{ /\\ } \dots \text{ /\\ } P_m$$

where S_i is a HOL term denoting a set (in HOL set theory) and $P_1 \dots P_m$ are boolean terms constituting the schema's predicate. The S_i can have different HOL types. The mathematical toolkit of Z is represented by appropriate constant definitions based on HOL's set theory. Schema references as imports and sets, and schema expressions are represented by expansion into the representation above; hence their structure is "parsed away". Pretty-printing suppresses the printing of the expanded schemas — this helps the eye, but not the prover.

In Kraan and Baumann's representation "Z-in-Isabelle" [KB 95] a schema *A* is represented as a *theory* in Isabelle and the variables in the declaration part as *constants*:

```
AA = Toolkit +
consts      x1 ::  "S1_t"
            ...
            xn ::  "Sn_t"
translations
            "A" == " [ x1 : S1; ...; xn : Sn | P1 ...Pm]"
```

Here the S_{i}_t are the types of the S_i which are terms denoting a set in an independent set-theory based on an implementation of the sequent calculus LK that is contained in the Isabelle distribution. The interpretation of [..|..] is similar to the one of "Z in HOL": all parameters are conjoined. The consequence of making the signatures of schemas globally visible is that all schema expressions have to be expanded by the parser.

Compiling schemas to (PVS)-theories, the work of [ES 94] is conceptually rather close to "Z-in-Isabelle". However, due to the introduction of intermediate constants for predicative parts, it comes very close to our ideal of a structure preserving encoding — except that lists of equations have to be generated for variables stemming from different Z schema declarations (like x_1 in B). The compilation cannot support schema-as-sets. Deduction at the level of the schema-calculus is possible in a very restricted way only. As a consequence of the lacking re-translation (pretty-printing), intermediate stages in theorem proving cannot easily be reinterpreted as schemas.

ICL's ProofPower [Jon 92] is a commercial product based on a deep encoding. Schemas are represented as sets of bindings, and schema operations work on these sets. More information on this work can be found in [BG 94].

The theorem prover Ergo [RS 93] implements an untyped meta-logic in Prolog on top of which Zermelo-Fränkel set theory and a theory for Z are encoded. The basic proof mechanism of Ergo is window inference which is augmented by a tactic language. Because of the lack of an underlying type system, many "typing" subgoals arise during proofs. Automatically invoked tactics called *elves* are used to try and proof these subgoals. If the elves fail, the user has to tackle these subgoals interactively.

Like Ergo, the implementation of Z in EVES [Saa 92, MS 95] incorporates the mathematical toolkit of Z and uses theorems about it as rewrite rules. The basic proof commands of EVES are specialised on syntactic categories of formulas, e.g. "eliminate a quantifier". Proven theorems are automatically used by more complex rewriting and simplification procedures.

Jigsa$\mathcal{W}$ [Mar 94] is a deep encoding done in 2OBJ for $\mathcal{W}$, a logic proposed for Z [WB 92], which it faithfully encodes. Since there are no meta-variables in 2OBJ, general theorems about Z cannot be expressed schematically. Proof procedures that produce proofs for each instance of the schematic theorem have to be coded instead. Reasoning in a deep encoding in this way reduces performance considerably. Furthermore, Maharaj has encoded Z in type-theory using LEGO [Mah 90].

There are also direct implementations for dialects of Z, e.g. Balzac [Har 91, Jor 91] and CADi$\mathbb{Z}$ [TH 95] allowing at least for single step inferences. CADi$\mathbb{Z}$ is based on a sequent calculus that is applied using a proof procedure called "Gentzen". It also incorporates a decision procedure for integers, and is currently being extended with a tactic language.

1.3 Overview of this Paper

This paper proceeds as follows: TZN identifies the following hierarchy of syntactic categories: *expressions, predicates, schema-expressions, paragraphs* and *theories*. In Section 2, we present our encoding bottom-up, successively giving a semantic representation for each of these categories and demonstrate its consequences for Z. In Section 3, we discuss proof support for our encoding "Z in Isabelle/HOL".

2 Representing Z in Isabelle/HOL

2.1 Conformance with TZN

For any embedding of a logic, the question of a "faithful encoding of one calculus in another" has to be raised. The question is moot here, since a Z calculus is not available now: the draft definition of a calculus in Appendix F of TZN is very rudimentary.

Our embedding *conforms* to Z in the sense of TZN (p. 2):

> A specification conforms to the standard for the Z notation, if and only if the formal text is written in accordance with the syntax rules and is well-typed.
>
> A deductive system for Z conforms to the standard, if and only if its rules are sound with respect to the semantics.

The core of TZN consists of the definition of two *partial* functions $\langle\!\langle\ \rangle\!\rangle^{\tau}$ and $\langle\!\langle\ \rangle\!\rangle^{M}$ that assign to an element of each syntactic category a *type* and a *value (meaning)*. Therefore a calculus conforms to the standard if it reflects the semantic function *where it is defined*. The semantic functions are interpreted in an untyped universe of Zermelo-Fränkel set theory. Thus, in the semantic universe, objects like $\{0, \{0\}\}$ may occur that are illegal in the typed set theory of HOL. This does not mean that $\{0, \{0\}\}$ is legal in Z; in fact, one of the major objectives of $\langle\!\langle\ \rangle\!\rangle^{\tau}$ is to rule out such expressions.

The following figure outlines the semantic situation: Let HOL_{τ} be the set of HOL-terms of type τ. Moreover, let Z_{τ} denote the set of Z-expressions of a type τ, and ZF the class of sets in ZF-theory into which all elements of Z_{τ} are mapped. Since the universe into which Z is interpreted is a ZF-set that is closed under cartesian products and $\mathbb{P}$ ([Nic 95],pp.17) — technically speaking it is a set of ordinality ω^{ω}—, this universe is isomorphic to the set in which HOL is interpreted via the semantic function described in [GM 93]: the set closed under function construction $A \rightarrow B$ and the type bool. The parser discribed in this paper maps all terms in Z_{τ} to specific HOL_{τ}-terms, such that the diagram below commutes up to isomorphism for all τ.

$$\begin{tikzcd}
Z \arrow[rr, "\langle\!\langle\ \rangle\!\rangle^{M}\text{[Nic 95]}"] \arrow[dr, "\text{Z-Parser}"'] & \cong & ZF \\
& HOL \arrow[ur, "\text{Sem [GM 93]}"'] &
\end{tikzcd}$$

This meta-proof shows that the encoding of the typed set-theory (for which the parser is the idenitity) conforms to the draft standard. It is perhaps surprising to discover that the semantic basis of Z as described in the rather compact mathematical notation in TZN is just an equivalent to the standard model of the typed λ-calculus. It remains to show how the syntactical, "notational" facet of Z can be handled by our *Z-Parser.*

2.2 Syntactic Issues

Previous Z encodings cope differently with the question of Z syntax. Most of them try to encompass the LaTeX presentation of the Z syntax, e.g. [BG 94, KB 95]. Since lexical issues are very important for both presentation and readability, we decided to remain close to the Z syntax as presented in TZN. The draft standard proposes several ASCII-based representations of Z. The interchange format is based on SGML, while the email format is primarily conceived as a human readable lightweight interchange format, rather than for processing by tools. In the email format the character % is used to flag special strings, to disambiguate them from others.

Concerning the lexical representation, our encoding supports the email format. The following table gives an impression of how our representation relates to the various lexical representations:

Z operator	Email Format	Z in Isa/HOL	Meaning
$\fatsemi$	%;	%;	compose
$\circ$	%o	%o	functional compose
$\lhd$	<:	<:	domain restrict
$\rightarrowtail\!\!\!\twoheadrightarrow$	>-->>	>-->>	bijection

Below we present the exceptions made necessary in order to cope with HOL types:

Z operator	Email Format	Z in Isa/HOL	Meaning
:	:	::	type membership
$\in$	%e	:	set containment

Parsing is done at two different levels, interleaved by a conversion phase. For the e-mail format level, we introduce a theory file Zproto.thy providing the necessary syntax and translation rules. The second level is the semantic representation level. ML-functions convert the abstract syntax parsed by the e-mail format level to the latter. A short excerpt of Zproto.thy is shown below:

```
Zproto = HOL +
syntax
...
Schema::[DeclParts, bool] => Zschema
"_sch3"::[Name, DeclParts] => Zschema                    ("+--_/---_/---")
"_sch4"::[Name, DeclParts, Predicate] => Zschema         ("+--_/---_/|--_---")
...
"_sch7"::[Name,Formals,DeclParts,Predicate]=>Zschema    ("+--_[_]/---_/|--_---")
translations
"+-- N ---  D ---"                == " N = Schema D True"
"+-- N ---  D |-- P ---"          == "(N = (Schema D P)"
"+-- N [M] ---   D |-- P ---"     == "(N = (%M. Schema D P)"
```

In Isabelle, syntactic sorts like `Name` or `DeclParts` can be introduced that are treated as non-terminals of a grammar (adapted from TZN), while constant declarations like `_sch3` can have the character of a grammar rule, with the pragma `("+--_/---_/---")` introducing alternative mixfix syntax where the `/` informs the pretty-printer where to place optional linebreaks. Translation rules may successively transform such raw-syntax-trees into trees where only symbols occur for which semantic information is available. Schema is such a function that takes a declaration and a predicate and returns a Z schema (of the type Zschema).

According to Zproto.thy, the example of schema A can be presented as follows:

```
"+-- A ---
      x1:S1;
      ...;
      xn:Sn
 |--  P1 &
      ... &
      Pm
 ---"
```

All schemas are parsed according to the signature of Zproto.thy. After computing its signature, a schema is translated to the semantic level, where the right binders with the right scope are generated. The pretty-printing will, according to a user-controlled variable `pretty_level`, partially or completely perform the retranslation. In the sequel, we focus on the description of the semantic representation level.

2.3 Expressions

The predicate logic of Z uses the usual connectives and quantifiers with their standard semantics. We therefore model Z predicates as Boolean functions P::'a => bool. Consequently, we can directly map the logical connectives of Z to the ones provided by HOL. Representing the set theory of Z is similarly simple, since Z is strongly typed. In particular, all elements of a set must have a common type. It follows that there is a universal set for each type that contains all elements of that type. This is why there is no need to explicitly refer to types in the Z language. From these observations, we conclude that the set theory of HOL is convenient to represent Z sets. The set constructor 'a set models the powerset operator $\mathbb{P}$ of Z at the level of types, and the set operations of Z directly translate to the corresponding operations of HOL.

Primitive Operations and the Mathematical Toolkit. The mathematical toolkit is introduced as an Isabelle theory ZMathTool. It is a conservative extension of some theories from the HOL library just as all constants of the toolkit in TZN are *defined* by the core language of Z:

```
ZMathTool = Finite + Integ + Arith +
types ('a,'b) "<=>" = "('a*'b) set"        (infixr 20)
```

A pivotal type in the mathematical toolkit is the *relation* which is defined as an infix type constructor <=>. This type will be used to shape all sorts of function spaces.

According to our lexical principles, we are now able to present the toolkit as a suite of constant definitions (the technique is equivalent to [BG 94]). On the right-hand side of the type definition some parsing information is given together with the binding values.

```
consts
...
rel            ::"['a set, 'b set] => ('a <=> 'b) set"   ("_ <-->_"      [5,4]4)
partial_func   ::"['a set,'b set] => ('a <=> 'b) set"    ("_ -|-> _"     [5,4]4)
total_func     ::"['a set,'b set] => ('a <=> 'b) set"    ("_ ---> _"     [5,4]4)
partial_inj    ::"['a set,'b set] => ('a <=> 'b) set"    ("_ >-|-> _"    [5,4]4)
total_inj      ::"['a set,'b set] => ('a <=> 'b) set"    ("_ >--> _"     [5,4]4)
...
```

```
defs
...
rel_def           "A <--> B  == Pow (A %x B)"
partial_fun_def   "S -|-> R  == {f. f:(S<-->R) & (ALL x: y1 y2.
                                        (x,y1):f  &  (x,y2):f-->(y1=y2))}"
total_func_def    "S ---> R  == {f. f:(S -|-> R) & (dom f) = S }"
partial_inj_def   "S >-|-> R == {f. f:(S -|-> R) &
                                (ALL x1:dom f x2:dom f.
                                     ((f %^ x1)=(f %^ x2)) --> (x1 = x2))}"
total_inj_def     "S >--> R  == (S >-|-> R) Int (S ---> R)"
end
```

Conformance of the toolkit with TZN is easy to verify: Just compare the definitions in ZMathTool to the ones in TZN. Furthermore, the laws given in [Spi 92] can be derived as theorems from ZMathTool. Especially the theorem:

$$\bigcap_{x:\{\}} P(x) = \{y.\ true\}$$

holds, in contrast to ZF where the result of this intersection over the empty index set is defined equal to {} because there are no universal sets in this untyped theory. In *typed* set theories like in Z or in HOL, the complement of a set is always defined.

Set Expressions, Function Application and Abstraction: On the basis of ZMathTool, it is straight-forward to represent declarations by membership predicates, for example the declaration of a partial injective function *f*:

```
f : N >-|-> N
```

Here ":" is the usual set membership operator of HOL set theory and N is the set of all natural numbers. The type of *f*, $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$ is modelled by

```
f :: nat set <=> nat set
```

This type is automatically inferred from the predicate. As a consequence of modelling functions as binary relations, we need a new application operator to apply f

```
"%^":: ['a set <=> 'b set, 'a] => 'b
```

which is defined by the Hilbert operator (as in [BG 94]):

```
f %^ x == (@y. (x,y) : f)
```

This treatment of partiality again conforms to TZN, where this extension of the semantic function is explicitly justified (p. 36):

> An example is the definition of application: for example, in function application, when the argument is outside the domain of the function, then no meaning is explicitly given. Different interpretations of Z can ascribe different meanings to an ill-formed function application.

There is a lively debate on the issue of partiality in the Z community and in the specification community in general. TZN's approach of using partial semantic functions allows designers of proof support systems to resolve issues like partiality the way it suits them best. Although counter-intuitive propositions like 1/0 = 1/0 are indeed provable by reflexivity in our encoding — in contrast, e.g., to [KB 95] — we believe that our model of function application considerably simplifies deduction while still conforming to the semantics of Z. Since the semantics of Z is partial, specifiers cannot rely on any semantics of function applications outside their domain. Reasoning about expressions like "1/0" cannot provide them with any information about their specification. On the other hand, modelling the partiality of the semantics function, e.g. by making "%^" partial, would only complicate deduction.

Abstraction is defined analogously[2]::

```
consts      LAM :: ['a => bool, 'a => 'b] => 'a <=> 'b set
defs        LAM S E == { (x,y) . S x & y = E x }
```

The general type of schemas which are introduced in the next section is 'a => bool. Thus LAM takes a schema S and an expression E in the signature of S and returns the desired relation.

Schema Expressions. The core problem of shallow Z-embeddings is the treatment of the declarations defined in a schema. They are called the *signature* of a schema and is computed in TZN via the already mentioned partial function $\langle\!\langle\ \rangle\!\rangle^{\tau}$. Our example *B* of schemas as imports demonstrates the delicate fusion of a schema's signature with the signature of its context: identifiers with equal *names* in the declaration part of the imported schema *A* and in the surrounding declarations of *B* are identified

Our approach suggests making this dependency explicit and considering a schema *S* as a boolean-valued function on the sub-signature of the context. It is the task of the parser to keep track internally of the signature of schemas and schema expressions. It must hence substitute a schema references *S* in a schema context by an application *S x*, where *x* represents the variables of the surrounding context that have to be identified with the ones declared in *S*. Note that the construction of signatures will always assure that the signature of the context will contain the elements of *x*.

However, the type-scheme

$$\tau = \alpha_1 \to \ldots \to \alpha_n \to \text{bool}$$

for the representation of a schema is inappropriate because, in some places, we need an Isabelle type-scheme that subsumes all types of schema representations. This is necessary to represent schemas-as-sets and to deal with expressions of the schema calculus, as well. For this reason, we use the uncurried version of the type scheme above:

$$\tau' = \alpha_1 \times \ldots \times \alpha_n \to \text{bool}$$

This type-scheme can be generalised to the type $\alpha \to \text{bool}$. This means that schemas are basically represented as predicates over tuples of variables in our encoding.

The fundamental mechanism to define representations of schemas are *schema binders* that are defined as follows[3]:

```
consts      SB0  :: "('a => bool) => ('a => bool)"             (binder "SB0" 10)
            SB   :: "(['a, 'b] => bool) => (('a * 'b) => bool)"  (binder "SB" 10)
defs        SB0_def  "SB0 P == P"
            SB_def   "SB P    == (% (x,y). (P x y))"
```

[2]Since functional abstraction is not defined in the current version of TZN, we use the definition of [Spi 92] to justify the conformity of our encoding.

[3]In our implementation, the situation is slightly more complex for purely syntactic reasons.

The pragma (binder "SB" 10) tells Isabelle to treat SB like a quantifier and provide it with additional syntax. The term:

$\mathsf{SB}\ x_1.\ \ldots\ \mathsf{SB}\ x_{n-1}.\ \mathsf{SB0}\ x_n.\ \mathsf{P}\ x_1 \ldots x_n$

has the type τ' iff P has the type τ. We call such a term a *schema lifter* over P of *length* n.

Schema lifters are intended to be suppressed by the pretty-printer. The conversion-phase within the parser generates lifters that are lexically sorted because schemas with permutated declarations are semantically identical.

Simple Schemas. Schema binders only model the binding structure of schemas. Simple declarations in Z not only declare a new variable but also contain membership constraints. These are reflected by the schema declaration DECL. DECL corresponds to the operator symbol Schema of the syntax (see section 2.2) and replaces it during the conversion phase of the parser.

```
consts      DECL :: [ bool, bool ] => bool      (" _ |--- _ " ...)
defs        DECL_def   "P |--- Q    == P & Q"
```

This enables us to represent a schema

```
┌── S ────────────
│   x : ℕ
│   y : ℤ
│   f : ℤ ⤖ ℤ
├──────────────
│   P x f y
└────────────────
```

in Isabelle as:

```
consts      S:: "int <=> int * int * int => bool".
def         S == SB f. SB x. SB0 y. x : ℕ  & y : ℤ & f: ℤ >-->ℤ   |---  P x f y
```

Semantically, a simple schema is just the conjunction of the two parameters. The new constructor DECL is needed to reflect the structure of schemas in the representation. Thus, the pretty-printer can reproduce the concrete syntax of a schema. In particular, the *user-defined* ordering of the declarations in a schema and the distinction between membership propositions in declarations and predicates can be reconstructed. The schema binders are not printed at all, and the sorting of declarations remains internal. However, the parser has to store the lexical names of the variables in the signature so the pretty-printer can hide the effects of α-conversions on schema binders.

Generic Schemas. The representation of simple schemas naturally extends to the case of *generic* schemas by abstracting the body schema *S* in the form: $\lambda M::\alpha$ *set. S.* The generic schema

```
┌── P[R] ────────────
│   x : R ⇸ ℤ
├──────────────
│   T x
└──────────────────
```

is converted by the parser to

```
consts      P:: "'a set => ( 'a set <=> 'int set ) => bool"
defs        P == % R . SB0 x .  x : R -|-> ℤ  |---  T x
```

The polymorphism of `P` in `'a` models exactly the type constraints of Z that are checked by type-checkers like *fuzz*. How the parameter set `R` is instantiated depends on the particular application context: if `R` is not instantiated explicitly, the universal set `{x :: 'a. true}` over the parameter *type* `'a` can be used.

Schema-as-import, Schema-as-sets. We recall the structures *B* and *C* from the introduction:

```
__ B ____________          __ C __________
| A                        |  y : A
| x_1 : T                  |  c : T
|______________            |______________
| true                     |  Q(y.x_1) (y.x_2)(c)
|__________________        |__________________
```

hat are consequently represented by:

```
defs
      B == SB x_1. ... SB0 x_n. A(x_1,...,x_n) & x_1 :  S_1  |---  true
      C == SB y. SB0 c.  y : {z. A z} & c : T |--- Q (fst y) ((fst o snd) y) (c)
```

Schema Expressions. The expressions of the schema calculus closely resemble predicate logic formulas. There are the usual connectives as well as universal and existential quantification, and, as one expects, their intuitive semantics can be based on the understanding of predicate logic.

The construction of schema lifters mimics the construction of signatures of schemas in TZN. Hence, logical connectives and quantifiers can be lifted to the corresponding schema expressions using an appropriate schema lifter.

In the case of the schema connectives, the signature of schema expressions is the union of the signatures' operands if the variables in the intersection of the signatures have identical types. Otherwise, the schema expression is ill-formed. For example, the schema-conjunction $N \cong S \wedge R$, where the signature of S is (s, u) and the one of R is (t, u, v) is represented by the following lifter over the conjunction:

```
      defs "N == SB s t u. SB0 v.  (S (s,u) & (R (t,u,v))"
```

As with simple schemas, the pretty-printer can decide on the basis of the predicate following the schema lifters which schema expression is represented, and print it accordingly.

2.4 Z Paragraphs and Sections

The formal text of a Z specification consists of a sequence of paragraphs which gradually introduce the given types, global variables, and schemas of the specification. The paragraphs forming a specification can be organised as a collection of named sections. Each paragraph can augment the environment by declarations of constants or schemas.

In our encoding, Z paragraphs and sections are represented as Isabelle theories. The way new schemas are introduced, by the key word `defs`, ensures a conservative construction of these theories. This guarantees that the representation process of Z into Isabelle does not introduce inconsistencies.

3 Proof Support

Since our encoding preserves the structure of specifications, two possibilities for structured reasoning about specifications arise. First, we can combine theorems about single schemas in a controlled way, and second, we are able to lift theorems of predicate logic to the level of schema expressions, and thus come to a truly mechanised schema *calculus*.

3.1 Structured Theories about Z Specifications

Complex Z specifications are usually structured as follows: first, axiomatic declarations introduce the constants which the rest of the specification is based upon (the "model" of the specification). Afterwards, simple schemas provide the basic notions relevant for the system to specify. Finally these are combined, possibly in several steps, to schemas that provide "the" system specification. For sequential systems, for example, a common approach is to combine several schemas describing "substates" of the system to the final state schema, and to provide schemas specifying different aspects of an operation, e.g. various cases of normal behaviour and behaviour in error cases, that are combined, using the schema calculus, to a schema specifying "the" operation.

Since we can refer to schemas as first-class objects in our encoding, we have an elegant way to form "local theories" about single schemas and augment schema references with the information provided by these theories. A theorem P about schema A can be expressed as an implication of the Isabelle meta-logic

$A\ (x_1,...,x_n)$ ==> $P\ x_1\ ...\ x_n$

This theorem can be used to extend the predicate of

B == SB x_1. ... SB0 x_n. $A(x_1,...,x_n)$ & x_1 : S_1 |--- true

Another possibility to reason about schemas is to successively transform them into equivalent representations by equational reasoning. One starts with an equation whose right-hand side is a meta-variable

$A\ (x_1,...,x_n) = ?X$

Using transitivity of equality and Isabelle's simplifier or other suitable tactics one can now gradually instantiate the metavariable ?X and prove

$A\ (x_1,...,x_n) = A_{simp}$

where A_{simp} is a simplified version of A. This equation can now be used — possibly again using the simplifier — to replace references to A in other schemas by A_{simp}.

Given the explicit representation of schema references, these transformations are technically simple. Still, they provide the possibility to develop the theory of a specification in "layers", starting at the layer of the mathematical toolkit and the axiomatic declarations, and ascending via simple schemas to the more complex schema expressions that finally make up a system description.

First experiences with reasoning about a non-trivial Z specification show the practical advantage of our structure preserving encoding. We have applied the technique described above to transform operation schemas of a specification of a simplified embedded controller into disjunctive normal form and used the simplifier —

instantiated with rules about the mathematical toolkit — to eliminate unsatisfiable disjuncts from the normal form. This procedure is part of an approach to generate test cases from model-based specifications [DF 93].

The major practical problem here is the performance of the prover when faced with large subgoals. With encodings that do not preserve the structure of specifications, all schema references in the operation schema to transform are inevitably expanded. This is neither appropriate nor practically feasible: on one hand, one will want to control expansion of definitions to come to sensible test cases. On the other hand, each additional literal in the schema's predicate increases computation time and space requirements of the prover because the distributivity laws duplicate subformulas and context-dependent simplification is mandatory to find unsatisfiable disjuncts.

We computed the normal form of our example operation schema in a bottom-up fashion: first computing the normal form of the state schema and of the operation schema without expanding schema references, and second combining the results and computing the normal form of the combination. Simplification helps a lot to control combinatorial explosion: more than two thirds of the state schema's disjuncts can be reduced to *false*, and only 8 of 44 disjuncts of the combined schema are satisfiable and form the final outcome of the computation. Each disjunct consists of about 20 literals. Just expanding schema references and compute the normal form in one step is absolutely infeasible: the algorithm did not terminate in an over-night run!

3.2 Lifting Predicate Logic to Schema Expressions

A technically more demanding possibility arising from our encoding of schema expressions is to lift theorems of predicate logic to the corresponding schema expressions. The basic idea is to use Isabelle's lifting of meta-variables that occurs when resolving a theorem with the matrix of the (meta-)universal quantifier "!!".

Consider what happens when resolving the extensionality theorem ext

```
!! x. ?f x = ?g x ==> ?f = ?g
```

with a theorem of predicate logic, say and_commute

```
?A & ?B = ?B & ?A
```

Forward chaining (and_commute RS ext) results in

```
%x. ?A x & ?B x = %x. ?B x & ?A x
```

This means the meta-variables ?A and ?B of and_commute are lifted to *functions* in x. We can use this mechanism to lift predicate logic theorems to *schema* binders using the theorems SB0_ext and SB_ext:

```
(!!x. f x = g x) ==> (SBinder0 f) = (SBinder0 g)
(!!x. f x = g x) ==> (SBinder f)   = (SBinder g)
```

Combining these two in an ML function lift

```
fun lift th 0 = th
   | lift th 1 = th RS SB0_ext
   | lift th n = (lift th (n-1)) RS SB_ext;
```

allows us to lift a predicate logic theorem to a theorem about schemas with signatures of arbitrary length, e.g.

```
lift and_commute 3;
val it =     "(SB x xa. SB0 xb. ?A x xa xb & ?B x xa xb) =
              (SB x xa. SB0 xb. ?B x xa xb & ?A x xa xb)" : thm
```

This lifting mechanism enables us to reason *directly* about schema expressions simply by reusing predicate logic theorems. In particular, it is not necessary to — implicitly or explicitly — expand the schemas of a particular expression and reason at the level of their (combined) predicates. Because the length of schemas can always be inferred by auxilliary tactical functions it is possible to hide schema lifting from the user's view by providing versions of Isabelle's resolution functions like RS or rtac that implicitly lift predicates to schemas of appropriate length.

4 Conclusion

We have presented a shallow, TZN-conforming encoding of Z into higher-order logic that nevertheless preserves the structure of specifications. Moreover, our representation allows deductions at a structural level, i.e. in the schema calculus. Other approaches to proof assistants for Z either implement a proof tool from scratch and accept the disadvantages in terms of implementation work, lack of reuse, and error-proneness to come up with a tool tailor-made for Z, or they encode Z into a logical framework like HOL or ZF. The latter approaches usually choose a deep encoding if they want to deal with the schema calculus, or they sacrifice the structure of Z specifications and represent only "flattened" specifications where all schema references are expanded. Providing a shallow encoding that still allows us to deal with schemas at the logical level is the major contribution of our work.

There is a price to pay for a shallow embedding: we cannot represent all aspects of the semantics of Z in logical terms, some have to be dealt with at the level of syntax. We chose to put the dividing line between syntax and logic at exactly the point of the Z semantics where it gets a very "syntactic" flavour, i.e. where the signatures of schemas and the binding structure induced by schema references are concerned. The mechanisms needed here can easily be understood as manipulations of sets of identifiers, and can hence safely be implemented in a parser by introducing appropriate schema binders "SB".

An advantage of this approach is that we, unlike deep embeddings like ProofPower [Jon 92], need not deal with "syntactical" issues in the logic. This also includes the issue of type checking. Since Z types are handled by the Isabelle parser, we do not need to reason about them explicitly. Untyped provers like Ergo [RS 93] must provide specialised tactics to prove type constraints. A first experiment based on the ZF encoding of Isabelle has shown that the type constraints provided by HOL greatly enhance efficiency of deduction. Similar proofs of (simple) propositions about Z schemas as predicates need much more search in ZF than in HOL (we usually had to use best_tac instead of fast_tac to find a proof).

Depending on the context of their use, schemas can have three different interpretations in Z: as sets, in schema calculus, and as predicates. We picked the latter as the basis of our embedding because schemas are often used in predicative context and proof engines like Isabelle are tuned to deal with predicates most efficiently. This choice lead to the idea of schema lifters and consequently enabled us to come up with a representation of the schema calculus which, to our knowledge, is the first in a shallow embedding.

As did Bowen and Gordon [BG 94], we map the set theory of Z to the one of HOL. Z is strongly typed and the apparent similarities to the HOL set theory are much greater than to other set theories like, e.g., ZF. We believe this and the encoding of the schema calculus in higher-order abstract syntax justifies our claim that our embedding conforms to the Z draft standard.

4.1 Future Work

Many specifications using Z heavily depend on the mathematical toolkit, and consequently many subgoals arising while reasoning about Z specifications are basically propositions about the toolkit. Such "standard" subgoals may be provable automatically by specialised proof procedures that are based on an extensive collection of theorems about the toolkit. The generic proof infrastructure of Isabelle, notably the highly customisable simplifier, provides a means to implement such proof procedures. We are currently investigating what degree of automation can be reached, in particular for specifications stemming from specific application domains.

For the time being, we have concentrated on representing the most crucial features of Z — this subset should be augmented and combined with other formal methods as envisaged in [Kri$^+$95]. The powerful pretty-printer of Isabelle provides much potential to support other syntactic representation like the *TZN interchange format*, LaTeX or Tcl/Tk for the interactive vizualization of mathematical, high-quality notation. In this way, the elaborate theorem proving facilities of Isabelle are made available to support practical work with Z such as analyses of specifications, transformational development, and test case generation and evaluation on the basis of Z. Refining the theorem proving support for testing based on formal specifications will be one major focus of activities in the project ESPRESS (see also [Jäh$^+$95]).

Acknowledgement. We thank the referees for constructive comments which helped improving the presentation. Thanks also for pointing us to [ES 95].

References

BG 94] Bowen, J. P., Gordon, M. J. C.: Z and HOL. In Bowen, J.P. and Hall, J.A. (ed.): *Z Users Workshop*, Cambridge 1994, Workshops in Computing, pp. 141-167, Springer Verlag, 1994

[DF 93] Dick, J., Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In Woodcock, Larsen (eds.), Proc. Formal Methods Europe, pp. 268-284, LNCS 670, Springer Verlag, 1993.

[EM 85] Ehrig, H. Mahr, B.: *Fundamentals of Algebraic Specification: Volume 1: Equations and Initial Semantics*, Springer Verlag, 1985

[ES 95] M. Engel, J.U.Skakkebæk: Applying PVS to Z. ProCoS II document [ID/DTU ME 3/1], Technical University of Denmark. 1995.

[GM 93] Gordon, M.J.C., Melham, T.M.: *Introduction to HOL: a Theorem Proving Environment for Higher order Logics*, Cambridge University Press, 1993.

[Har 91] Harwood, W. T.: Proof rules for Balzac. Technical Report WTH/P7/001, Imperial Software Technology, Cambridge, UK, 1991.

[Jäh+95] S.Jähnichen (director): ESPRESS — Engineering of safety - critical embedded systems. Online information available via http://www.first.gmd.de/org/espres.html.

[Jon 92] Jones, R. B.: ICL ProofPrower. BCS FACS FACTS Series III, 1(1):10-13, Winter 1992.

[Jor 91] Jordan, L. E.: The Z Syntax Supported by Balzac II/1. Technical Report LEJ/S1/001. Imperial Software Technology, Cambridge, UK, 1991.

[KB 95] Kraan, I., Baumann, P.: Implementing Z in Isabelle. In Bowen, Hinchey (eds.), ZUM '95: The Z Formal Specification Notation, pp. 355-373, LNCS 967, Springer Verlag, 1995.

[Kri$^+$95] Krieg-Brückner, B., Peleska, J., Olderog, E.-R., Balzer, D., Baer, A.: Uniform Workbench — Universelle Entwicklungsumgebung für formale Methoden. Technischer Bericht 8/95, Universität Bremen, 1995. Also available online via http://www.informatik.uni-bremen.de/~uniform.

[KSW 96] Kolyang, Santen, T., Wolff, B: Correct and User-Friendly Implementations of Transformation Systems. Proc. Formal Methods Europe, Oxford. LNCS 1051, Springer Verlag, 1996.

[Mah 90] Maharaj, S.: Implementing Z in LEGO. Unpublished M.Sc.thesis. Departement of Computer Science, University of Edinburgh, September 1990.

[Mar 94] Martin, A.: Machine-Assisted Theorem-Proving for Software Engineering, Unpublished PhD Thesis, University of Oxford, 1994.

[MS 95] Meisels, I., Saaltink, M.Z.: The Z/EVES Reference Manual (draft). Technical report TR-95-5493-03, ORA Canada, December 1995

[Nic 95] Nicholls, J. (*ed.*, prepared by the members of the Z Standards Panel): Z - Notation. Version 1.2. ISO-Draft. Online: http://www.comlab.ox.ac.uk /oucl/users/andrew.martin/zstandard/.14th September 1995.

[Pau 94] Paulson, L. C.: *Isabelle - A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994.

[RS 93] Robinson, P.J., Staples, J.: Formalizing a Hierarchical Structure of Practical Mathematical Reasoning. *Journal of Logic and Computation* 3 (1), pp. 47-61, 1993

[Saa 92] Saaltink, M.Z.: Z and EVES. In Nicholls, J.E. (ed.) Z User Workshop, York 1991, Workshops in Computing, pages 223- 242. Springer Verlag 1992

[Spi 92] Spivey, J.M. : *The Z Notation: A Reference Manual* (2nd Edition). Prentice Hall, 1992.

[Spi 92a] Spivey, J.M.: The *f*uzz Manual, Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK 2nd edition, 1992

[TH 95] Toyn, I., Hall, J.: Proving Conjectures using CADi$\mathbb{Z}$. York Software Engineering Ltd., September 1995.

[WB 92] Woodcock, J.C.P., Brien, S.M.: $\mathcal{W}$: A logic for Z. In Nicholls, J.E. (ed.) *Z User Workshop*, York 1991, Workshops in Computing, pp. 77-96. Springer Verlag 1992