Department of Computer Science
University of Essex, England
Technical Report CSM-344

Two Semantic Embeddings of Z Schemas in Isabelle/HOL

Norbert Völker

October 2, 2001

Abstract

This report investigates two semantic embeddings of Z schemas in Isabelle/HOL. The first represents Z values as elements of a type class with polymorphic type constructors and overloaded operators. In contrast, the second embedding uses a Z universe: all Z values are represented as elements of a single monomorphic HOL type.

1 Introduction

Embedding a specification formalism such as Z [Spi88] in higher order logic (HOL) is attractive because of the automation provided by theorem prover tools like the HOL system [GM93] or Isabelle/HOL [Pau93]. In [Völ01b], a deep Isabelle/HOL embedding of the logic Z_C [HR00] was constructed. Z_C is a first-order logic which has been proposed as a foundation for Z schemas. The embedding keeps Z_C completely separated from HOL: Z_C propositions and terms are represented as members of a new HOL type. While this clean separation is attractive from a purist point of view, the explicit representation of syntactical aspects such as substitution means that proof goals can get cluttered with relatively low level syntactic side conditions. It also allows for very little reuse of the existing HOL theories and tools. This motivates the search for more shallow, semantic embeddings which permit a greater reuse of the extensive Isabelle/HOL libraries.

In agreeance with most previous work [JG94, KSW96] on the embedding of Z schemas in HOL, both semantic embeddings in this report will identify Z_C formulae with HOL formulae. This means that there is no separate representation of Z_C logical values and logical operators. Instead the standard HOL type *bool* and logical connectives are used. Similarly,

Z_C variables and operations such as substitution will be identified directly with their HOL counterparts. Thus, there is no need to reason explicitly about substitution or freeness of variables.

The challenge of embedding Z schemas in HOL is that a direct identification of Z types with corresponding HOL type constructors is not possible. The problem is caused by schemas which require a generic record type where field values can be elements of different HOL types. Such a type can not be constructed in the simply typed polymorphic λ-calculus underlying HOL. Previous work [JG94, KSW96] on Z embeddings has sidestepped this problem by treating field labels outside of HOL: schemas are encoded externally as tuples before analysing them in HOL. This results in shallow embeddings well suited for reasoning about concrete specifications. Because of the encoding, rules such as commutativity of schema disjunction can not be derived within HOL. As noted in [JG94], this can cause a certain amount of duplication in proof efforts. Furthermore, it makes these embeddings less suitable for an analysis of the schema calculus itself.

This report investigate two approaches which are more faithful in the sense that field labels are preserved in the HOL representation of schemas. The underlying view of Z schemas is essentially that of [HR00]. In contrast to that source and most of the Z literature, we will speak of "records" instead of "bindings".

The report only provides a summary of the actual Isabelle/HOL theory development. The reader is refered to the source files [Völ01a] for further details.

2 Labels

A concrete representation of labels as strings has been chosen here. Strings are simply lists of characters in Isabelle/HOL.

$$string \quad = \quad char\ list$$

In the Z notation, labels can be adorned by suffixing them with special characters, namely '!' (shriek), '?' (query) and '$\prime$' (prime).

$$\begin{array}{lcl} special\_chars & :: & char\ set \\ special\_chars & = & \{\ \text{'!'},\ \text{'?'},\ \text{'}\prime\text{'}\} \end{array}$$

The *suffix* of a string is defined to be the tail starting with the occurrence of the first special character. If there are no special characters in a string, then the suffix is the empty list.

$$\begin{array}{lcl} suffix & :: & string \rightarrow string \\ suffix\ s & = & dropWhile\ (\lambda c.\ c \notin special\_chars)\ s \end{array}$$

Not all strings are deemed to be legal Z labels. Instead, it is required that the suffix of a string contains no duplicate characters, is sorted and only consists of special characters.

$$\begin{array}{lcl} legal\ s & = & let\ x\ =\ suffix\ s\ in \\ & & nodups\ x\ \wedge\ sorted\ (op\ \leq)\ x\ \wedge\ set\ x \subseteq special\_chars \end{array}$$

In order to allow reuse of the constant name "legal", it is overloaded on a new type class *label_class* which comprises all list types and hence also the type of strings:

$$\begin{array}{lcl} \texttt{axclass}\ \mathit{label\_class} & < & \mathit{term} \\ \texttt{instance}\ \mathit{list} & :: & (\mathit{term})\ \mathit{label\_class} \\ \mathit{legal} & :: & (\alpha :: \mathit{label\_class}) \rightarrow \mathit{bool} \end{array}$$

Adorning a label with a special character is modelled by corresponding operations. Again, in order to allow a reuse of constant names, overloaded constants are chosen:

$$\mathit{shriek}, \mathit{query}, \mathit{prime} :: (\alpha :: \mathit{label\_class}) \rightarrow \alpha$$

For legal strings, each of these operations toggles the occurrence of the corresponding special character in the suffix. For example, if a legal string s is already "primed", i.e. it contains the prime character in its suffix, then *prime* s is the string with this prime character removed. All three operations are idempotent on the set of legal strings and preserve legality:

$$\begin{array}{lcl} \mathit{legal}\ (s :: \mathit{string}) & \Rightarrow & \mathit{legal}\ (\mathit{shriek}\ s) \wedge \mathit{shriek}\ (\mathit{shriek}\ s) = s \\ \mathit{legal}\ (s :: \mathit{string}) & \Rightarrow & \mathit{legal}\ (\mathit{query}\ s) \wedge \mathit{query}\ (\mathit{query}\ s) = s \\ \mathit{legal}\ (s :: \mathit{string}) & \Rightarrow & \mathit{legal}\ (\mathit{prime}\ s) \wedge \mathit{prime}\ (\mathit{prime}\ s) = s \end{array}$$

It should be noted that there are alternative ways to model labels in HOL. For example, it would be possible to define a datatype which separates the stem of a label from the special character suffix. One could also cater for different encodings of labels by introducing an axiomatic class with rules that capture the properties of the operations prime, query and shriek. The treatment outlined above has the advantage that it corresponds directly to concrete labels in Z schemas.

3 Records in HOL

Because HOL is based on the simply-typed lambda calculus, there is no way to have a generic HOL "record type constructor" which allows field values of arbitrary, different Z types. There are a number of methods one can try to overcome this problem:

1. Suppression of labels by extra-logical encodings
2. Introduction of a new type constructor and new constants for every new record type
3. Extension of HOL's type system by dependent record types.
4. Representation of records in a class of HOL types and using overloaded record operations.
5. Encoding field values as elements of a single type and then using a HOL representation of type-homogenous records, i.e. records whose field values are all of the same type.

The first method underlies previous Z embeddings [JG94, KSW96] and has already been discussed above. The second method has been implemented in Isabelle/HOL[NW98] and HOL98. Its drawbacks are that each new record type has to be declared explicitly and that there is no HOL representation of generic record operations such as extension, restriction and extraction. Although implemented to some extend in PVS[RSC95], the problem with the third method is that it is still not clear how best to combine records and polymorphism [GJ96]. Since type extension to the core of a theorem prover is a serious task, it makes sense to wait until the theoretical issues have been sufficiently clarified.

The fourth method underlies the class/overloading approach to the embedding of Z schemas in Section 8. The last method is used indirectly in the universe approach in Section 10.

4 Homogenous Records as Labelled Lists

While the representation of general records in HOL is problematic, there are several ways to encode type-homogenous records where all fields values are of the same type. Here an encoding as lists of pairs will be chosen:

$$\alpha\ llst \quad = \quad (string \times \alpha)\ list$$

The representation of a type-homogenous record r with field values of some type α in $(\alpha\ llst)$ is the labelled list obtained by sorting the set of fields of r in lexicographic order according to the field names. A list element(l, x) corresponds to a record field with label l and value x. For example, the representation of the empty record is the empty list, while the representation of record $\langle\!|\, x = 1, y = 2 \,|\!\rangle$ is the list $[(\text{“}x\text{”}, 1), (\text{“}y\text{”}, 2)]$. The representation is unique because of the sorting of list elements by their labels.

In [Völ01b], type-homogenous records were represented as a new type $(\alpha\ rcd)$ isomorphic to the set of all finite mappings from strings to α. That type has the advantage of a 1-1 correspondence between records and their HOL representation. However, the drawback is that the new type constructor *rcd* is not a HOL datatype. This means that types containing an occurrence of *rcd* are not covered by Isabelle/HOL's packages for datatypes and primitive recursive functions [Pau94]. This would entail a lot of extra work in the theory development. It is for this technical reason that the encoding of records as labelled lists was chosen here.

Not all elements of type $(\alpha\ llst)$ correspond to records. In fact, a list represents a record if it is sorted by labels and its labels are unique:

$$\begin{array}{lcl} is\_rcd & :: & \alpha\ llst \ \rightarrow\ bool \\ is\_rcd\ x & = & sorted\ (\lambda x\, y.\ fst\ x \leq fst\ y)\ x\ \wedge\ nodups\,(map\ fst\ x) \end{array}$$

The constant *labels* is overloaded to allow reuse. For an element x of type $(\alpha\ llst)$, it is defined as the *fst*-image of the set underlying x.

$$\begin{array}{lcl} labels & :: & (\alpha :: label\_class) \ \rightarrow\ string\ set \\ labels\ (x :: \alpha\ llst) & = & fst\ `\ (set\ x) \end{array}$$

A fundamental record operation is the retrieval of the value associated with a label l. The operation returns some unspecified value ($SOME\ y.False$) if l does not occur among the labels of the list. The operation is defined by primitive recursion over lists:

$$\begin{array}{lcl} lkp\_llst\ [\,]\ l & = & (SOME\ y.\ False) \\ lkp\_llst\ ((a,b)\ \#\ xs)\ l & = & if\ a = l\ then\ b\ else\ lkp\_llst\ xs\ l \end{array}$$

The binary union of records and the restriction of a record to a set of labels are modelled via functions *un_llst* and *restr_llst*:

$$\begin{array}{lcl} un\_llst & :: & [\alpha\ llst, \alpha\ llst] \to \alpha\ llst \\ un\_llst\ x\ y & = & insort\ (\lambda x\ y.\ fst\ x \le fst\ y) \\ & & \quad (x\ @\ [a \in y.\ a \notin set\ x]) \\ restr\_llst & :: & [\alpha\ llst, string\ set] \to \alpha\ llst \\ restr\_llst\ x\ ls & = & [p \in x.\ fst\ p \in ls] \end{array}$$

Function ($insort\ R$) sorts a list according to some relation R while the infix operator "@" appends two lists. The list comprehension $[a \in x.\ P\ x]$ filters the elements fulfilling a predicate P from a list x.

Compatibility of two records ensures that their union is again a record:

$$\begin{array}{lcl} compatible & :: & [\alpha\ llst, \alpha\ llst] \to bool \\ compatible\ x\ y & = & (\forall a\ b\ c.\ (a,b) \in set\ x \wedge (a,c) \in set\ y \\ & & \quad \Rightarrow b = c) \\ \multicolumn{3}{l}{is\_rcd\ x \wedge is\_rcd\ y \wedge compatible\ x\ y \Rightarrow is\_rcd\ (un\_llst\ x\ y)} \end{array}$$

The reader is referred to the Isabelle source files for the further development of this theory.

5 A Class of Records

The representation of records in HOL will be based on a datatype of labelled pairs:

$$\texttt{datatype}\quad \alpha\ \beta\ lprd \quad = \quad Lpair\ string\ \alpha\ \beta$$

The empty record $\langle\!|\ |\!\rangle$ is represented by the single element ( ) of the one-element type *unit*. Non-empty records are constructed in HOL starting from ( ) by successively adding fields using the constructor *Lpair*. As an example, here is the representation of a record with two fields a and b associated with values $1 :: \mathbb{N}$ and $True :: bool$:

$$\langle\!|\ \text{“}a\text{”} = 1, \text{“}b\text{”} = True\ |\!\rangle \ \tilde{=}\ Lpair\ \text{“}a\text{”}\ 1\ (Lpair\ \text{“}b\text{”}\ True\ ())$$

In this way, every record with field values of HOL class *term* can be represented as an element of a type in class *rc*:

$$\begin{array}{lcl} \texttt{axclass}\ rc & < & label\_class \\ \texttt{instance}\ unit & :: & (term)\ rc \\ \texttt{instance}\ lprd & :: & (term, rc)\ rc \end{array}$$

The pretty-printing facilities of Isabelle make it possible to use the usual notation for concrete records. Internally, this is translated into the corresponding HOL terms. To aid legibility, this paper will also use the normal record notation for concrete elements of class *rc*.

The set of labels of a record is defined by recursion over the type structure of class *rc*:

$$\begin{array}{lcl} \mathit{labels}\ (\,) & = & \{\,\} \\ \mathit{labels}\ (\mathit{Lpair}\ s\ a\ x) & = & \{s\}\ \cup\ \mathit{labels}\ x \end{array}$$

The above representation of records in HOL and the definition of the *labels*-function appear simple and natural. Unfortunately, the problem with this representation becomes clear when trying to define the basic generic record operations *lookup* (lookup of the value belonging to a label), *upd* (field update or extension) and *restr* (restricting a record to a set of labels).

For these functions, the dependency of the result type on the argument type can not be expressed in HOL. As a consequence, the result type has to be represented by a new type variable. This leads to the following type declarations:

$$\begin{array}{lcll} \mathit{lookup} & :: & [\alpha :: \mathit{rc}, \mathit{string}] & \rightarrow\ \beta \\ \mathit{upd} & :: & [\alpha :: \mathit{rc}, \mathit{string}, \beta] & \rightarrow\ (\gamma :: \mathit{rc}) \\ \mathit{restr} & :: & [\alpha :: \mathit{rc}, \mathit{string}\ \mathit{set}] & \rightarrow\ (\beta :: \mathit{rc}) \end{array}$$

In the HOL theorem proving tradition, the uncontrolled introduction of axioms is shunned whenever possible. This is because of the grave danger of creating an inconsistent logic. Instead, new constants are usually introduced by applying definition principles which are guaranteed to be safe. Unfortunately, Isabelle/HOL's definition facilities do not cater for functions whose result type contains type variables not occurring in the argument types. Hence it is necessary to introduce non-definitional axioms which characterise the three overloaded record operations.

`rules`

$$\begin{array}{lcl} s = t & \Rightarrow & \mathit{lookup}\ (\mathit{Lpair}\ s\ a\ x)\ t = a \\ s \neq t & \Rightarrow & \mathit{lookup}\ (\mathit{Lpair}\ s\ a\ x)\ t = \mathit{lookup}\ x\ t \\ & & \\ & & \mathit{upd}\ (\,)\ s\ a = \mathit{Lpair}\ s\ a\ (\,) \\ s < t & \Rightarrow & \mathit{upd}\ (\mathit{Lpair}\ t\ b\ x)\ s\ a = \mathit{Lpair}\ s\ a\ (\mathit{Lpair}\ t\ b\ x) \\ s = t & \Rightarrow & \mathit{upd}\ (\mathit{Lpair}\ t\ b\ x)\ s\ a = \mathit{Lpair}\ s\ a\ x \\ s > t & \Rightarrow & \mathit{upd}\ (\mathit{Lpair}\ t\ b\ x)\ s\ a = \mathit{Lpair}\ t\ b\ (\mathit{upd}\ x\ s\ a) \\ & & \\ & & \mathit{restr}\ (\,)\ \mathit{ls} = (\,) \\ s \in \mathit{ls} & \Rightarrow & \mathit{restr}\ (\mathit{Lpair}\ s\ a\ x)\ \mathit{ls} = \mathit{Lpair}\ s\ a\ (\mathit{restr}\ x\ \mathit{ls}) \\ s \notin \mathit{ls} & \Rightarrow & \mathit{restr}\ (\mathit{Lpair}\ s\ a\ x)\ \mathit{ls} = \mathit{restr}\ x\ \mathit{ls} \end{array}$$

Why do these axioms not compromise consistency? We will argue the case of the operator *restr*, the safety of the other record operations follows similarly. Consider a HOL theory which includes the record class *rc* and the operator *restr* as defined above. The crucial observation is that by orienting the three *restr* equations from left to right, one obtains a terminating, orthogonal, and hence complete term rewriting system $\mathcal{R}$ [Klo92]. Let $=_{\mathcal{R}}$

be the equivalence relation generated by $\mathcal{R}$ on terms of class rc, i.e. relation $t =_{\mathcal{R}} t'$ holds provided the equality of t and t' can be proven from the three *restr* equations. Let Ter_0 denote the set of all terms of class rc without any occurrence of the *restr* operator. Since terms in Ter_0 can not be reduced further in $\mathcal{R}$, they are in normal form. By the uniqueness of normal forms, it follows that for each term $(t :: \alpha :: rc)$ there is at most one term $t' \in Ter_0$ such that $t =_{\mathcal{R}} t'$. This leads to the following requirement for the *restr* function:

$$restr\ x\ ls \quad = \quad y,\ if\ y =_{\mathcal{R}} restr\ x\ ls\ \wedge\ y \in Ter_0$$

By the remarks mentioned above, this specification provides an unambiguous meaning of $(restr\ x\ ls)$ provided this term can be rewritten in $\mathcal{R}$ to an element of Ter_0. No constraint will be put on the value of $(restr\ x\ ls)$ if such a rewrite is not possible.

By construction, this specification of the *restr* function is consistent with the three *restr* axioms. It leads to a theory extension which is definitional in the sense that the new axioms determine the meaning of terms by reducing them to terms in the original theory. This ensures that consistency is not compromised.

In order to model binary logical operations on Z schemas, it is necessary to introduce further an operation which appends two records. The safety of this definition follows from a similar argument as above.

$$\begin{array}{lll}
 & & append :: [\alpha :: rc, \beta :: rc] \rightarrow (\gamma :: rc) \\
 & & append\ ()\ y = y \\
 & & append\ x\ () = x \\
s < t & \Rightarrow & append\ (Lpair\ s\ a\ x)\ (Lpair\ t\ b\ y) \\
 & & = Lpair\ s\ a\ (append\ x\ (Lpair\ t\ b\ y)) \\
s = t & \Rightarrow & append\ (Lpair\ s\ a\ x)\ (Lpair\ t\ b\ y) \\
 & & = Lpair\ s\ a\ (append\ x\ y) \\
s > t & \Rightarrow & append\ (Lpair\ s\ a\ x)\ (Lpair\ t\ b\ y) \\
 & & = Lpair\ t\ b\ (append\ (Lpair\ s\ a\ x)\ y)
\end{array}$$

The reader might wonder why *append* was not defined using the two equations:

(1) $append\ x\ () = x$
(2) $append\ x\ (Lpair\ s\ a\ y) = upd\ (append\ x\ y)\ s\ a$

The problem lies in the second equation which contains a hidden type variable on the right-hand side. This is because the occurrence of the term $(append\ x\ y)$ can be of any type $\gamma :: rc$. Clearly such an axiom would be semantically problematic, consider for example the case of a non-empty record x and $\gamma = unit$. Note that the above argumentation about consistency breaks down because the resulting term rewriting system would not be orthogonal - rule (2') can be applied with different types of γ on the right-hand side.

The fact that the defining equations of the four record operations do not introduce new type variables makes them suitable for use in the Isabelle/HOL simplifier. In particular, the value of these functions for concrete records can be determined automatically. However, this will only

work provided the result type fits the type of the term as determined by its context. This means for example that simplification will automatically rewrite the term

$$(lookup\ \langle\!|\text{“}x\text{”} = 1|\!\rangle\ \text{“}x\text{”} :: \mathbb{N})$$

to 1 using the first *lookup* rule. On the other hand, there is no rule applicable to the term:

$$(lookup\ \langle\!|\text{“}x\text{”} = 1|\!\rangle\ \text{“}x\text{”} :: bool)$$

Similarly, one can prove automatically the statements:

$$\begin{array}{lcl} append\ \langle\!|\text{“}b\text{”} = 1|\!\rangle\ \langle\!|\text{“}c\text{”} = True|\!\rangle & = & \langle\!|\text{“}b\text{”} = 1,\ \text{“}c\text{”} = True|\!\rangle \\ append\ \langle\!|\text{“}b\text{”} = 1|\!\rangle\ \langle\!|\text{“}c\text{”} = 2|\!\rangle & \neq & \langle\!|\text{“}b\text{”} = 2,\ \text{“}c\text{”} = 1\ |\!\rangle \end{array}$$

because the type of the *append* term as determined by the context fits with the type expected by the *append* rewrite rules. On the other hand, it is impossible to use these equations to either prove or disprove the following statements:

$$\begin{array}{lcl} append\ \langle\!|\text{“}b\text{”} = 1|\!\rangle\ \langle\!|\text{“}c\text{”} = True|\!\rangle & = & \langle\!|\text{“}b\text{”} = True,\ \text{“}c\text{”} = 1|\!\rangle \\ append\ \langle\!|\text{“}b\text{”} = 1|\!\rangle\ \langle\!|\text{“}c\text{”} = True|\!\rangle & \neq & \langle\!|\text{“}c\text{”} = True\ |\!\rangle \end{array}$$

The failure to prove results about such “ill-formed” terms is in agreeance with the fact that no value was specified for record operator terms which can not be completely evaluated, see the definition of *restr* above.

6 Z Types

Following [HR00], we consider Z to be a typed set theory with natural numbers as the only primitive type. Type forming operations are sets, binary products and records. This type structure can be represented easily in HOL by a datatype *zt*:

$$zt \quad = \quad NatT \quad | \quad SetT\ zt \quad | \quad PrdT\ zt\ zt \quad | \quad RcdT\ (zt\ llst)$$

The restriction to these four type constructors is mainly for presentation purposes. The approach can be extended to accommodate further primitive types such as lists or partial functions.

The type *zt* as defined above contains elements which do not represent a Z type. The problem is caused by the fact that in the expression $(RcdT\ Ts)$, the list of pairs *Ts* might either not represent a record or it might contain illegal strings as labels. Predicate *legal* excludes such elements.

$$\begin{array}{lcl} legal\ NatT & = & True \\ legal\ (SetT\ A) & = & legal\ A \\ legal\ (PrdT\ A\ B) & = & legal\ A\ \wedge\ legal\ B \\ legal\ (RcdT\ As) & = & (\forall (s,T) \in set\ As.\ legal\ s \wedge legal\ T) \\ & & \quad \wedge\ is\_rcd\ As \end{array}$$

7 A Class of Z Values

The HOL representation of records as labelled products makes it possible to introduce a class zc which represents all Z values. Subclass zrc consists of the records in zc.

$$\begin{array}{l}
\texttt{axclass}\ zc\ <\ label_class \\
\quad \mathbb{N} \quad :: \quad zc \\
\quad \times \quad :: \quad (zc, zc)\ zc \\
\quad tset \quad :: \quad (zc)\ zc \\
\texttt{axclass}\ zrc < zc, rc \\
\quad unit \quad :: \quad zrc \\
\quad lprd \quad :: \quad (zc, zrc)\ zrc
\end{array}$$

The datatype $tset$ consists of sets labelled with their Z type.

$$\texttt{datatype}\quad \alpha\ tset \quad = \quad TSet\ zt\ (\alpha\ set)$$

The use of type constructor $tset$ instead of set in class zc makes it possible to distinguish empty sets of the same HOL type but different Z type.

Membership of a value $(x :: \alpha :: zc)$ in a Z type $(T :: zt)$ is denoted as $(x : T)$. This typing relationship is defined inductively over the structure of class rc.

$$\begin{array}{lcl}
(n :: \mathbb{N}) : T & = & (T = NatT) \\
(a, b) : T & = & (\exists\ A\ B.\ T = PrdT\ A\ B \wedge a : A \wedge b : B) \\
(TSet\ A\ x) : T & = & (A = T\ \wedge\ \forall\ a \in x.\ a : T) \\
(x :: unit) : T & = & (T = RcdT\ [\,]) \\
Lpair\ s\ a\ b : T & = & legal\ s\ \wedge\ (\forall\ l : labels\ b.\ s < l)\ \wedge \\
 & & (\exists\ A\ B.\ a : A\ \wedge\ b : RcdT\ B \\
 & & \quad \wedge\ T = RcdT\ ((s, A)\ \#\ B))
\end{array}$$

Note that Z typing is finer than HOL typing. While elements of the same Z type have to be of the same HOL type, the converse is in general not true due to the fact that labels of record fields effect the Z type but not the HOL type. Consider for example the records $zlblot\text{“}a\text{”} = 1\rangle\!\rangle$ and $\langle\!\langle\text{“}b\text{”} = 1\rangle\!\rangle$ - both have HOL type $(lprd\ \mathbb{N}\ unit)$ but their Z types are $RcdT\ [(\text{“}a\text{”}, NatT)]$ and $RcdT\ [(\text{“}b\text{”}, NatT)]$, respectively.

The principle Isabelle tool to prove properties of classes is the introduction of axiomatic type classes. This method was used to establish uniqueness of Z typing:

$$a : A\ \wedge\ a : B \quad \Rightarrow \quad A = B$$

Class zc contains elements which do not respresent well-formed Z values. Similar to the case of type zt, predicate $legal$ excludes such elements.

$$legal\ x \quad = \quad \exists\ T.\ x : T\ \wedge\ legal\ T$$

From the general definition, equations can be derived which characterise

the predicate *legal* by recursion over the structure of class *zc*:

$$\begin{array}{lcl} legal\ (n :: \mathbb{N}) & = & True \\ legal\ (x, y) & = & legal\ x\ \wedge\ legal\ y \\ legal\ (TSet\ T\ x) & = & legal\ T\ \wedge\ \forall a \in x.\ a : T \\ legal\ () & = & True \\ legal\ (Lpair\ s\ a\ x) & = & legal\ s\ \wedge\ legal\ a \wedge\ legal\ x \\ & & \quad \wedge\ (\forall l \in labels\ x.\ s < l) \end{array}$$

Another general rule which was proven with the help of axiomatic classes is:

$$a : RcdT\ Ts \quad \Rightarrow \quad labels\ a = labels\, Ts$$

Unfortunately, the rules in Isabelle's axiomatic type classes are restricted to a single type variable each. This makes it impossible to prove generic statements which contain more than one type variable of the underlying class. For example, it is not possible to prove a rule which relates the type of (*restr x ls*) to the type of x in general.

8 Z Schemas using Overloading

8.1 Representation of Schemas

Schemas are the distinguishing feature of the Z specification formalism. They consists of a declaration part and a predicate part. In HOL, a schema can be modelled by pairing a Z type record with a predicate:

$$\alpha\ schema \quad = \quad zt\ llst \times (\alpha\ \rightarrow bool)$$

The first component reflects the Z typing information contained in the declaration part. More precisely, the Z type of the elements belonging to a schema S is $RcdT\ (fst\ S)$.

The second component of a schema reflects the predicate part. The set of records associated with a schema consists of those records which are of the declared type and which fullfill the predicate.

$$\begin{array}{lcl} set\_of & :: & \alpha\ schema\ \rightarrow\ \alpha\ set \\ set\_of\ (Ts, P) & = & \{x.\ x : RcdT\ Ts\ \wedge\ P\ x\} \end{array}$$

A schema is said to be *normal* if its predicate only holds for records which are of the declared type:

$$is\_normal\ (Ts, P) \quad = \quad (\forall x.\ P\ x \Rightarrow x : RcdT\ Ts)$$

Obviously, any schema can be modified to a normal schema by setting the predicate part to *False* for arguments not of the right type. This modification does not change the set of elements associated with a schema:

$$\begin{array}{l} is\_normal\ (Ts, (\lambda x.\ P\ x\ \wedge\ x : RcdT\ Ts)) \\ set\_of\ (Ts, (\lambda x.\ P\ x\ \wedge\ x : RcdT\ Ts)) \quad = \quad set\_of\ (Ts, P) \end{array}$$

For the definition of schema operations below, it is convenient to introduce an inclusion operator which checks whether the restriction of a Z value to the type of a Z schema is contained in the associated set:

$$\begin{array}{lcl} op\ IN & :: & [\alpha, \beta\ schema] \rightarrow bool \\ x\ IN\ S & = & restr\ x\ (labels\ (fst\ S)) \in set\_of\ S \end{array}$$

8.2 Schema Operations

Using the above representation of schemas, it is straightforward to translate logical schema operations to HOL. Below, we will give a brief account of negation, disjunction and existential quantification. The other logical schema operations can be treated analogously.

Negation on schemas is defined by:

$$\begin{array}{lcl} NOT & :: & \alpha\ schema \rightarrow \alpha\ schema \\ NOT\ (Ts, P) & = & (Ts, (\lambda x.\ x : RcdT\ Ts \wedge \neg(P\ x))) \end{array}$$

The disjunction of two schemas is given by:

$$\begin{array}{l} op\ OR\ ::\ [\alpha\ schema, \beta\ schema] \rightarrow \gamma\ schema \\ S_1\ OR\ S_2\ = \\ \quad let\ Ts = un\_llst\ (fst\ S_1)\ (fst\ S_2)\ in \\ \quad\quad (Ts,\ (\lambda x.\ (x\ IN\ S_1 \vee x\ IN\ S_2) \wedge\ x : RcdT\ Ts)) \end{array}$$

Lastly, the definition of existential quantification over a label l in a schema S is:

$$\begin{array}{l} EXI\ ::\ [string, \alpha, \beta\ schema] \rightarrow \gamma\ schema \\ EXI\ l\ (A :: \alpha)\ (S :: \beta\ schema)\ = \\ \quad let\ Ts = restr\_llst\ (fst\ S)\ \{s.\ s \neq l\}\ in \\ \quad\quad (Ts,\ (\lambda x.\ (\exists\ l\ (a :: \alpha).\ (upd\ x\ l\ a\ : \beta)\ IN\ S)\ \wedge\ x : RcdT\ Ts)) \end{array}$$

The purpose of the argument A in this definition is purely to specify the HOL type of the element associated with label l in schema S. This avoids the introduction of a new type variable on the right-hand side of the equation.

The schemas produced by the three operators are per construction in normal form:

$$\begin{array}{l} is\_normal\ (NOT\ S) \\ is\_normal\ (S_1\ OR\ S_2) \\ is\_normal\ (EXI\ l\ T\ S) \end{array}$$

Here are some other basic algebraic properties of the schema operators which can be proven easily in HOL:

$$\begin{array}{lcl} is\_normal\ S & \Rightarrow & NOT\ (NOT\ S) = S \\ is\_rcd\ As \wedge is\_rcd\ Bs \wedge compatible\ As\ Bs & & \\ \quad\quad \Rightarrow & ((As, P)\ OR\ (Bs, Q)) = ((Bs, Q)\ OR\ (As, P)) & \\ is\_rcd As \wedge is\_normal\ (As, P) & & \\ \quad\quad \Rightarrow & ((As, P)\ OR\ (As, P)) = (As, P) & \end{array}$$

The class based approach outlined above can be used to reason about concrete schemas. As an example, the HOL theories [Völ01a] associated with this technical report provide two different definitions of a simple tank schema and then proceed to show their equivalence.

On the other hand, many generic schema calculus theorems can not be deduced in the class approach without introducing further non-definitional axioms about record operations and their effect on Z typing. This problem is caused mainly by the restriction of Isabelle's axiomatic classes to a single type variable. The need to introduce further axioms makes the class based approach less attractive for the theoretical investigation of schema calculi.

9 A Z Universe in HOL

9.1 Definition

As an alterative to the Z class approach sketched above, we abandon the idea of representing different Z types by corresponding HOL types. Instead, a single HOL type $\mathcal{Z}$ will be introduced which contains a representation of every Z value.

Such a Z universe could be constructed via an isomorphism with a subset of the disjoint sum over class *zc* [Völ99]. The exposition below provides an alternative axiomatic construction. As a first attempt, consider the definition of a HOL datatype:

$$\mathcal{Z} \stackrel{??}{=} Nat\ \mathbb{N} \mid Set\ zt\ (\mathcal{Z}\ set) \mid Prd\ \mathcal{Z}\ \mathcal{Z} \mid Rcd\ (\mathcal{Z}\ llst)$$

Unfortunately, there are two major problems with the constructor *Set*:

1. Datatype constructors are injective. This is not possible for *Set* since for any HOL type A the cardinality of type $(A\ set)$ is strictly greater than that of A.
2. Datatype constructors are total. For the constructor *Set*, this clashes with the fact that Z is a typed set theory where all elements of a set have to be of the same type.

These two problems suggest that $(Set\ T)$ should really be a partial function which is only defined for argument sets x whose elements are of Z type T. This partiality could be modelled explicitly in HOL by setting the target type of $(Set\ T)$ to be $(\mathcal{Z}\ opt)$. A more low cost approach is to keep $(Set\ T)$ a total function but to disregard argument sets x whose elements are not of Z type T. This is the method chosen here. Injectivity of function $(Set\ T)$ is only demanded for sets with elements of Z type T, thus eliminating the cardinality problem described above.

The axiomatisation of the universe $\mathcal{Z}$ will be based on an inductive construction of the carrier sets of Z types. For the definition of the carrier set of a record type, it is convenient to introduce two further functions on labelled lists. The first of these is the equivalent of the well-known *map* combinator on lists. The second is the the labelled list counterpart of the

product set ($\times$) and power set ($\mathbb{P}$) operators.

$$
\begin{array}{lcl}
\mathit{map\_llst} & :: & [\alpha \rightarrow \beta, \alpha\ \mathit{llst}] \rightarrow \beta\ \mathit{llst} \\
\mathit{map\_llst}\ f & = & \mathit{map}\ (\lambda\ (x, y).\ (x, f\ y)) \\
\mathit{llsts} & :: & \alpha\ \mathit{set}\ \mathit{llst} \rightarrow \alpha\ \mathit{llst}\ \mathit{set} \\
\mathit{llsts}\ [] & = & \{[]\} \\
\mathit{llsts}\ ((l, s) \#\ x) & = & \bigcup_{a \in s, y \in \mathit{llsts}\ x} \{(l, a) \#\ y\}
\end{array}
$$

The carrier function is defined by primitive recursion over type zt.

$$
\begin{array}{lcl}
\mathit{carrier} & :: & \mathit{zt} \rightarrow \mathcal{Z}\ \mathit{set} \\
\mathit{carrier}\ \mathit{NatT} & = & \mathit{Nat}'\ \mathit{nats} \\
\mathit{carrier}\ (\mathit{SetT}\ T\ x) & = & \mathit{Set}\ T'\ (\mathbb{P}\ (\mathit{carrier}\ T)) \\
\mathit{carrier}\ (\mathit{PrdT}\ A\ B) & = & \mathit{Prd}'\ (\mathit{carrier}\ A \times \mathit{carrier}\ B) \\
\mathit{carrier}\ (\mathit{RcdT}\ \mathit{Ts}) & = & \mathit{Rcd}'\ (\mathit{llsts}\ (\mathit{map\_llst}\ \mathit{carrier}\ \mathit{Ts}))
\end{array}
$$

Based on this definition, the Z universe can be specified by axioms which are a modification of the usual datatype axioms so as to deal with the *Set* constructor in the way outlined above. Firstly, there are the usual distinctness rules for the four constructors:

$$
\begin{array}{lcll}
\texttt{A-1} & \mathit{Nat}\ x & \neq & \mathit{Set}\ T\ y \\
 & \mathit{Nat}\ x & \neq & \mathit{Prd}\ y \\
 & \mathit{Nat}\ x & \neq & \mathit{Rcd}\ y \\
 & \mathit{Set}\ T\ x & \neq & \mathit{Prd}\ y \\
 & \mathit{Set}\ T\ x & \neq & \mathit{Rcd}\ y \\
 & \mathit{Prd}\ x & \neq & \mathit{Rcd}\ y
\end{array}
$$

Secondly, we have injectivity properties of the constructors. For the *Set* constructor, there are two rules. The first guarantees disjointness of the sets $(\mathit{carrier}\ (\mathit{Set}\ T))$ for $T :: zt$. The second rule requires injectivity of $(\mathit{Set}\ T)$ for arguments which are subsets of $(\mathit{carrier}\ T)$.

$$
\begin{array}{ll}
\texttt{A-2} & \mathit{inj}\ \mathit{Nat} \\
 & \mathit{Set}\ A\ x = \mathit{Set}\ B\ y \Rightarrow A = B \\
 & \mathit{inj\_on}\ \mathit{Set}\ (\mathit{sets}\ (\mathit{carrier}\ T)) \\
 & \mathit{inj}\ \mathit{Prd} \\
 & \mathit{inj}\ \mathit{Rcd}
\end{array}
$$

In case of HOL datatypes, the last axiom is an induction theorem which expresses the fact that every element can be written as a constructor term. The carrier function allows us to specify this property more succinctly:

$$\texttt{A-3}\quad \exists T.\ x \in \mathit{carrier}\ T$$

The soundness of these rules follows from the fact that the universe $\mathcal{Z}$ is isomorphic to the set $\{(x :: \alpha :: rc).\ \exists\ T.x : T\}$ of all typeable elements in class zc. A definitional construction of $\mathcal{Z}$ as a subset of the disjoint sum over rc is planned for the future.

9.2 Further Development

The definition of the carrier function and axiom `A-3` yield the following induction theorem:

$$\frac{\begin{array}{l}\forall n.\ P\ (Nat\ n)\\ \forall a\ b.\ P\ a \wedge\ P\ b\ \Rightarrow\ P\ (Prd\ (a,b))\\ \forall r.\ list\_all\ P\ (map\ snd\ r)\ \Rightarrow\ P\ (Rcd\ r)\\ \forall T\ xs.\ xs \subseteq carrier\ T \wedge\ xs \subseteq \{x.\ P\ x\}\ \Rightarrow\ P\ (Set\ T\ xs)\end{array}}{\forall x.\ P\ x}$$

The formulation of this theorem uses a function ($list\_all\ P$) which requires a predicate P to hold for all element of a list.

$$list\_all\ P\ x\ \ =\ \ \forall a \in set\ x.\ P\ a$$

The proof of the universe induction theorem is by induction over the Z type associated with each element of the universe by axiom `A-3`. The uniqueness of this associated Z type follows from the disjointness of carriers: type zt:

$$A \neq B\ \ \Rightarrow\ \ carrier\ A \cap carrier\ B = \{\}$$

Because of the existence and uniqueness of the associated Z type, a function *ztof* can be defined which associates each element of the universe with its Z type. The characteristic equation of this function is:

$$ztof\ x = T\ \ =\ \ (x \in carrier\ T)$$

The effect of function *ztof* on constructor terms is described by a system of primitive recursive equations:

$$\begin{array}{rlcl} & ztof\ (Nat\ n) & = & NatT\\ & ztof\ (Prd\ (a,b)) & = & PrdT\ (ztof\ a)\ (ztof\ b)\\ x \subseteq\ carrier\ T\ \Rightarrow & ztof\ (Set\ T\ x) & = & SetT\ T\\ & ztof\ (Rcd\ x) & = & RcdT\ (map\_llst\ ztof\ x)\end{array}$$

Inductive datatypes in HOL come with a package for the construction of primitive recursive functions solving such systems of equations. This package is based on a primitive recursion combinator. The distinctness and injectivity axioms of $\mathcal{Z}$ together with the induction theorem allow the definition of a primitive recursion combinator on $\mathcal{Z}$ as well. Its construction is entirely analogous to the case of an inductive datatype - the combinator can be defined as the graph of a suitable inductively defined relation. The main difference lies in the conditional guard to the equation describing the effect of a primitive recursive function on the *Set* constructor, see the *ztof* example above. The reader is referred to the source files for details of this development. The primitive recursion combinator provides a convenient means to define further functions on the universe.

10 Z Schemas using a Z Universe

In the universe approach, schemas and their operations are monomorphic. Furthermore, $\mathcal{Z}$ terms are build from the four universe constructors. Apart from these two differences, most definitions are completely analogous to the class approach. The type of schemas is:

$$schema \quad = \quad zt\ llst \times (zt \ \rightarrow \ bool)$$

In order to keep the number of occurences of the constructors *Rcd* and *RcdT* at bay, it pays off to introduce a Z typing operator which takes records (in the form of labelled lists) as arguments:

$$\begin{array}{lcl} op\ :_r & :: & [\mathcal{Z}\ llst, zt\ llst] \ \rightarrow \ bool \\ x :_r\ Ts & = & (ztof\ (Rcd\ x) = RcdT\ Ts) \end{array}$$

Here are the defining equations of the functions *is_normal* and *set_of* on the Z universe:

$$\begin{array}{lcl} set\_of & :: & schema \ \rightarrow \ \mathcal{Z}\ set \\ set\_of\ (Ts, P) & = & \{x.\ x :_r\ Ts \ \wedge \ P\ x\} \\ set\_of & :: & schema \ \rightarrow \ bool \\ is\_normal\ (Ts, P) & = & (\forall x.\ P\ x \Rightarrow x :_r\ Ts) \end{array}$$

The definition of the operator *IN* employs the labelled list restriction operator *restr_llst* instead of the overloaded record operator *restr*:

$$\begin{array}{lcl} op\ IN & :: & [\mathcal{Z}, schema] \ \rightarrow \ bool \\ x\ IN\ S & = & restr\_llst\ x\ (labels\ (fst\ S)) \in set\_of\ S \end{array}$$

The definition of the operators *NOT* and *OR* remains unchanged except for the omission of the *RcdT* constructor made possible by the definition of $:_r$ above:

$$\begin{array}{l} NOT\ (Ts, P) \ = \ (Ts, (\lambda x.\ x :_r\ Ts \wedge \neg(P\ x))) \\ S_1\ OR\ S_2 \ = \\ \quad let\ Ts = un\_llst\ (fst\ S_1)\ (fst\ S_2)\ in \\ \quad\quad (Ts, \ (\lambda x.\ (x\ IN\ S_1 \vee x\ IN\ S_2) \wedge \ x :_r\ Ts)) \end{array}$$

The definition of the *EXI* operator is simplified because there is no need to record type information by a spurious argument:

$$\begin{array}{l} EXI \ :: \ [string, schema] \ \rightarrow \ schema \\ EXI\ l\ S \ = \\ \quad let\ Ts = restr\_llst\ (fst\ S)\ \{s.\ s \neq l\}\ in \\ \quad\quad (Ts, \ (\lambda x.\ (\exists\ l\ a.\ un\_llst\ [(l, a)]\ x\ IN\ S \ \wedge \ x :_r\ Ts))) \end{array}$$

Compared to the class approach, the universe setting provides a superior framework for proving generic theorems about Z typing and the schema calculus. For example, the following rule is proven easily by ordinary list induction:

$$x :_r\ Ts \quad \Rightarrow \quad restr\_llst\ x\ ls :_r\ restr\_llst\ \ Ts\ ls$$

The Isabelle source files contain further schema calculus rules as well as a simple, concrete tank schema example.

11 Conclusions

Two semantic embeddings of Z schemas were constructed which allow a better reuse of the existing Isabelle/HOL framework compared to the deep embedding of Z_C in [Völ01b]. In both of these embeddings, HOL propositions are used directly to reason about schemas. Compared to previous work, the main difference lies in the fact that record labels are not treated outside of HOL but instead are explicitly modelled in the logic.

The two approaches differ in the representation of Z values. In the first embedding, Z types such as natural numbers, sets and products are mapped directly to corresponding HOL type constructors. The handling of Z schemas is based on a representation of records as polymorphic, labelled products. Record operators are overloaded and their definition requires non-definitional axioms. Consistency was established using careful reasoning based on term rewriting theory. The resulting framework appears to be a suitable foundation for the verification of concrete schemas. On the other hand, Isabelle's restriction to axiomatic classes with a single type variable makes it impossible to prove certain generic theorems about schemas in this setting. Because of this, the class method appears to be less suitable as a framework for the theoretical investigation of schema calculi.

For this purpose, it appears instead preferable to encode all Z values in a single HOL type (Z universe). The axiomatic definition of such a universe was carried out in analogy to the axiomatisation of an inductive HOL datatype. The requirement of type homogeneity for Z sets led to considering the universe constructor *Set* to be a partial function. While partial functions can be modelled explicity in HOL, a simpler approach was chosen here which leaves *Set* a total function but requires explicit reasoning about membership of arguments in carrier sets. Basic developments of the Z universe theory include proofs of an induction theorem, disjointness of carrier sets and the definition of a primitive-recursion combinator. The development of a schema calculus on the universe is analogous to the Z class approach but has the benefit that there is no need to introduce non-definitional axioms. Constants such as a primitive recursion combinator can be introduced definitionally and generic rules can be proven using normal Isabelle tactics. Reasoning about concrete Z schemas involves dealing with the universe constructors, but Isabelle's automated tactics makes handling these less awkward than one might expect.

Further work is planned on establishing a connection between the two embeddings in HOL. This should also lead to a more thorough justification of the universe axioms. Furthermore, as it stands, the work only deals with a minimal subset of the Z_C schema calculus. This needs be be extended to cover further operations such as schema piping. The suitability of the universe approach to modelling object-oriented formalisms merits further investigation.

Efficient reasoning about concrete schemas requires tuning of Isabelle's tactics. In particular, string handling is slow in Isabelle 99-2. This problem could be tackled either by implementing faster decision procedures or by choosing an alternative representation of labels.

References

[GJ96] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, UK, 1996.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[HR00] Henson and Reeves. Investigating Z. *JLC: Journal of Logic and Computation*, 10:43–73, 2000.

[JG94] J.P.Bowen and M.J.C. Gordon. Z and HOL. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 141–167. Springer-Verlag, 1994.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol 2*, pages 1–116. Oxford University Press, 1992.

[KSW96] Kolyang, T. Santen, and B. Wolff. A Structure Preserving Encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, LNCS 1125, pages 283–298. Springer Verlag, 1996.

[NW98] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. C. Newey, editors, *Theorem proving in higher order logics: TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Pau93] L.C. Paulson. Isabelle's object-logics. Technical Report 286, Computer Laboratory, University of Cambridge, 1993.

[Pau94] L.C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Automated Deduction — CADE-12*, LNAI 814, pages 148–161. Springer-Verlag, 1994.

[RSC95] J. Rushby and D.W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995.

[Spi88] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988.

[Völ99] N. Völker. Disjoint Sums over Type Classes in HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings of TPHOLs'99, Nice, France*, LNCS 1690, pages 5–18, 1999.

[Völ01a] N. Völker. Isabelle/HOL 99-2 files for "Two Semantic Embeddings of Z Schemas in HOL", 2001. On-line at `http://cswww.essex.ac.uk/Research/FSS/projects/c-z.html`.

[Völ01b] Norbert Völker. A Deep Embedding of ZC in Isabelle/HOL. Technical Report CSM-343, University of Essex, UK, 2001.