Encoding Z-style Schemas in Type Theory

Savi Maharaj

LFCS, Department of Computer Science, University of Edinburgh
JCMB, The King's Buildings, Edinburgh EH9 3JZ.
email: Savi.Maharaj@dcs.ed.ac.uk

Abstract. A distinctive feature of the Z specification language is its Schema Calculus which allows specifications to be packaged and put together to form new specifications. We investigate methods of transporting the Schema Calculus to the type theory UTT. We first attempt a direct encoding of schemas as Σ-types. This turns out to be unsatisfactory because encoding the operations of the Schema Calculus requires the ability to perform computations on the syntax of schemas, so we develop methods in which this syntax is also represented. These methods also depend upon Σ-types but use them in an unconventional fashion. We define a notion of implementation of a schema and use the LEGO proof-checker to prove some theorems about the interaction between implementations and our encodings of the operations of the Z Schema Calculus.

1 Introduction

The Z language [2, 5] is a formal notation based on set theory which provides an expressive, unambiguous language for writing specifications of programs. One of its main strengths is a structuring mechanism called the Schema Calculus[5] which allows specification modules to be put together in various ways to build new specifications. However there are elements which could be added to Z in order to make the language the basis of a complete program development methodology. For instance it is necessary to define what it means to implement a Z specification. It would also be helpful for proof support systems for Z to enable users to exploit the logical properties of the Schema Calculus itself.

The Unifying Theory of dependent Types (UTT) [9], and its implementation in the LEGO proof-checker [10], can also be viewed as a system for formal software development. Like set theory UTT can be used as a language for writing specifications. As a constructive type theory it also provides us with a notion of "program" which we can use to define an implementation. The LEGO proof-checker allows us to carry out machine-checked proofs in UTT. However UTT lacks some of the user-friendly properties of the specification languages used in industry.

In this paper we focus our attention on the structuring mechanism provided by Z, the Schema Calculus, which we attempt to interpret independently of the set theory within which it is used to structure theories. We investigate ways of transferring the Schema Calculus onto UTT as a base language instead of set theory. This is done by encoding a portion of the Schema Calculus in UTT. Our encoding enriches UTT by providing a way of writing structured specifications in UTT in the style of Z. This work can also be viewed as an alternative semantics for the Schema Calculus, based on type theory rather than set theory. This makes the Schema Calculus more independent than was originally envisaged by the designers of Z while preserving its essential features.

The encoding enables us to use LEGO to explore the properties of the operations of the Schema Calculus and their interaction with our definition of implementation. One of our aims is to gather a library of LEGO proofs of general properties of the Schema Calculus operations which we can re-use for proving properties of specifications or doing program verification.

For instance, the Schema Calculus provides an operation whereby two schemas S and T can be conjoined to produce a new schema $S \wedge T$. We would like to know whether if we have an implementation of schema $S \wedge T$ we can always derive from it implementations of S and of T — a kind of elimination rule. Conversely, if a program implements both S and T, or is an extension of implementations of these, does it then yield an implementation of $S \wedge T$? — a kind of introduction rule.

In Section 2 we introduce a small fragment of the Schema Calculus notation, just enough to illustrate the work that we have done. Then in Section 3 we discuss the ways in which we have tried to represent the Schema Calculus in LEGO, showing the problems that have motivated the development of each subsequent technique. We give the definitions that we finally settle upon and describe some results that we've proved. In Section 4 we give an example of the kind of reasoning that is made possible by my encoding. In Section 5 we discuss the relationship between our work and other semantic accounts of the Schema Calculus. We also compare our encoding with systems providing proof support for Z. Finally we discuss our conclusions in Section 6.

2 The Z notation

The Z notation consists of a core language based on set theory, and a structuring mechanism consisting of modules called *schemas* which can be combined in various ways using operations that make up the *Schema Calculus*.

Here is an example of a simple Z schema:

S

$x : \mathbb{N}$
$y : \mathit{list}\ \mathbb{N}$

$x \leq \mathit{length}\ y$

The schema consists of a *signature* and a *predicate*.

One of the operations of the Schema Calculus is schema conjunction; this involves *join*ing the signatures of two schemas and conjoining their predicates to give a new predicate over the joined signature. When two signatures are joined occurrences of the same identifier in both signatures are identified with each other.

Example:

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\ \hline
x : \mathbb{N} \\
z : list\ \mathbb{N} \\ \hline
x \geq length\ z \\ \hline
\end{array}$$

Conjoining S and T gives the schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{S \wedge T} \\ \hline
x : \mathbb{N} \\
y : list\ \mathbb{N} \\
z : list\ \mathbb{N} \\ \hline
(x \leq length\ y) \wedge (x \geq length\ z) \\ \hline
\end{array}$$

Other operations, which we won't describe here, include disjunction, implication, negation, inclusion of one schema within another, composition of schemas, and piping. Most of these make use of the *join* operation to combine signatures.

3 Representing the Schema Calculus in UTT

The Schema Calculus is the most distinctive feature of Z — it is what makes Z superior to using just set theory as a specification language. As a mechanism for structuring specifications it has proved its worth since the Z language is widely used in industry. We think it is rewarding to study this system independently of the set theory which it is used to structure, and in particular, to investigate whether this system may be usefully transported to another "core language" — that of constructive type theory.[1]

It must be pointed out that in standard Z the Schema Calculus and core language are interdependent, since it is possible to use schemas as types. We diverge from Z in isolating the Schema Calculus. The relationship between our work and standard Z will be discussed in Section 5.

For the purposes of our encoding, we first select a part of UTT to be our core language. This includes a selection of inductive types such as *nat*, *bool* and polymorphic *list*. See [10] and [9, 7] for discussions of, respectively, the practical and the theoretical aspects of introducing such inductive datatypes into the LEGO implementation of UTT. We take our definitions of these types and of basic functions that act upon them from the LEGO library [6]. The library functions and lemmas that we use are described in Appendix B.

[1] In previous work [11] we explored ways of "implementing" Z in LEGO by encoding set theory.

For many inductive types it is possible to define a computational (*bool*-valued) equality predicate on that type. We use the symbol $\stackrel{bool}{=}$ to denote such equality predicates. There are various propositional (*Prop*-valued) equalities which can be defined in UTT; of these we use Paulin-Mohring's inductive equality [3] which we denote by the symbol $=$.

Having defined a core language, we then introduce a new type representing schemas. Most of this section is concerned with the search for a suitable UTT type for this purpose. The type chosen should allow us to define all the operations of the Schema Calculus. Our intention is that UTT will play a double role in the encoding — it will act as the object language in which schemas are given meaning, but it will also be the metalanguage in which the Schema Calculus operations are defined, thereby enabling us to prove theorems about these operations *within* LEGO.

We make use of the fact that we are working in a constructive type theory to define a notion of program and an *implementation* relation between programs and schemas. Again we explore various possibilities for these definitions. Having made a final choice, we use LEGO to prove some theorems about the interaction between the implementation relation and the Schema Calculus operations.

Method 1: Schemas as $\boldsymbol{\Sigma}$ types We can use Σ-types to represent schemas in a manner similar to the way type theory is used in [8]. Then the schema S presented above is encoded in UTT as:

$$S\_sig \stackrel{def}{=} nat \times list\ nat$$
$$S\_pred \stackrel{def}{=} \lambda\, str : S\_sig.\, (\pi_1 str) \leq length\ (\pi_2 str)$$
$$S \stackrel{def}{=} \Sigma\, str : S\_sig.\, S\_pred\ str$$

This allows us to define an implementation: an implementation of S is just an object whose type is S, that is, a program of type S_sig paired with a proof that this satisfies the predicate S_pred. But how do we go about defining the operations of the Schema Calculus? In order to implement the *join* operation we need to compare the identifiers and types used in our schemas, that is, we need a computational equality on identifiers and on types. But with schemas as Σ-types we have no representation for the identifiers in schemas. Though we do have a representation of types (as UTT types) we cannot define a computational equality on these.

It may appear that our problem with identifiers would be solved by adding record types to the type theory. In fact this is not so, since the problem is not the absence of labels for the components of the Σ-type, but the impossibility of treating such labels as terms in the type theory that can be acted upon by functions defined in the type theory. What we need to do therefore is to encode the *syntax* of Z identifiers and types as terms in UTT that we can use to do computations.

Method 2: Syntactic names and types in signatures We introduce two new inductive types to represent identifiers and type names. The identifiers will be derived from the specification that is being encoded. For our example we have:

Definition 1 Ident.

$$\begin{array}{l} \mathit{Ident} \ ::= \ `x' \ | \ `y' \ | \ `z' \\ : \mathit{Type} \end{array}$$

We also introduce a new inductive type of Z type names. Here, $Given\_Type$ is an inductive type consisting of type names over which a specification is parameterised.

Definition 2 Ztype.

$$\begin{array}{lll} \mathit{Ztype} & ::= & natT \mid boolT \mid givenT \ Given\_Type \mid \\ & & funT \ (Ztype, Ztype) \mid prodT \ (Ztype, Ztype) \\ : \mathit{Type} & & \end{array}$$

We define a signature as a list of pairs of these syntactic identifiers and types:

Definition 3 Signature.

$$Signature \stackrel{def}{=} list \ (Ident \times Ztype) : Type$$

Since we still want to be able to relate these specifications to programs written in UTT, we must define the relationship between these syntactic signatures and types in UTT. To do this we define a semantic function, Typ that maps these syntactic types to UTT types: e.g. $Typ \ natT = nat$. Then we extend this to a function $Typify$ of type $Signature \rightarrow Type$[2] which forms a product of the types obtained by applying Typ to all of the $Ztype$s in a given signature, together with the unit type in the case of the empty signature. This essentially allows us to recapture our previous definition of a signature as a product of types.

We can then define a schema as consisting of a syntactic signature sig paired with a predicate over the semantic signature $Typify \ sig$:

Old definition 1 (Schema)

$$\begin{array}{l} Schema \stackrel{def}{=} \Sigma \, sig : Signature. \, (Typify \ sig) \rightarrow Prop \\ : Type \end{array}$$

We give the names $.sig$ and $.pred$, respectively, to the first and second projections from a $Schema$.

We define an implementation of such a schema $(sig, pred)$ as a program of type $Typify \ sig$ paired with a proof that it satisfies the predicate $pred$. This closely resembles the notion of implementation in Method 1.

[2] Actually UTT has a countable hierarchy of type universes. We select a particular universe, which, for simplicity, we refer to as $Type$.

Old definition 2 (Implementation)

$$Implementation \stackrel{def}{=} \lambda\, S : Schema.\ \Sigma\, str : Typify\ S.sig.\ S.pred\ str$$
$$: Schema \rightarrow Type$$

Implementations of a schemas S are objects of type ($Implementation\ S$).

In order to work with these syntactic functions we need to define several functions. One of these is *lookup* which, when given an identifier and a tuple *str* of type *Typify sig* for some signature *sig*, attempts to locate the identifier in *sig* and then projects the value in the corresponding position in *str*. Failure is handled by returning the value *void*. Typechecking will detect when *lookup* fails since the value returned in the case of failure will be of the wrong type. For example, in the following definition of the schema S, if we remove the identifier 'x' from *S_sig* (or if we paired it with some other type), then the definition of *S_pred* will fail to typecheck.[3]

The main use of *lookup* is in writing schema predicates. This is illustrated by the new definition of the schema S.

$$S\_sig \stackrel{def}{=} [(`x\text{'}, natT), (`y\text{'}, listT\ natT)]$$
$$S\_pred \stackrel{def}{=} \lambda\, str : Typify\ S\_sig.\ (lookup\ `x\text{'}\ str) \leq length\ (lookup\ `y\text{'}\ str)$$
$$S \stackrel{def}{=} (S\_sig, S\_pred)$$

Another example of a schema is the trivial, unsatisfiable, *Absurd_schema* which consists of the empty signature and the predicate $\lambda\, str : Typify\ [\,].\ absurd$.

Now we can begin to define the operations of the Schema Calculus. Consider the operation of conjoining two schemas S and S'. First we must form a new signature *newsig* by joining the signatures of S and S'. These two signatures may be inconsistent, in that there may be an identifier which occurs in both of them that is paired with different *Ztype*s in each occurrence, so we must find some way of handling this possibility. Next we must form a new predicate over the semantic signature *Typify newsig* by conjoining the two old predicates. To do this we have to make use of coercions which map programs in the new semantic signature back to programs in *Typify sig* and $Typify\ sig'$ since these types are the domains of the old predicates.

First we define the *join* function. This takes two signatures *sig* and sig' yields a new signature *newsig* together with a coercion back from *newsig* to *sig*. The coercion takes a tuple of type *Typify newsig* and produces a tuple of type *Typify sig* by projecting only those components that correspond to identifiers present in *sig*. No coercion back to the second signature sig' is computed since this may not exist if *sig* and sig' happen to be inconsistent.

Another operation *coerce* attempts to find coercions between arbitrary signatures. The type of *coerce* reflects the fact that it is partial: if no coercion exists

[3] This is accomplished by defining *lookup* in terms of another function *lookup_aux* which returns a dependent pair consisting of a type and a value of that type. We then use the second projection of this result.

it returns $in2\ void$.

$$coerce : \Pi\, S, S' : Signature.\, ((Typify\ S) \rightarrow (Typify\ S')) + unit$$

Now we can define schema conjunction. To conjoin S and S' we first *join* their signatures to form a new signature $newsig$. Then we attempt to coerce $newsig$ back to the signature of S'. This will fail if S and S' happen to be inconsistent, in which case we return $Absurd\_schema$ as our result. Otherwise we return a schema made up of $newsig$ and the predicates of S and S' conjoined and composed with coercions as appropriate.

The definition is as follows. We use square brackets here to enclose local definitions of identifiers, a useful feature of LEGO syntax. The function *case* is an elimination rule for the sum datatype: when applied to any $f : s \rightarrow u$ and $g : t \rightarrow u$, where s, t, and u are types, *case* returns a function of type $(s + t) \rightarrow u$.

Old definition 3 (And_schema)

$$\begin{array}{l}
And_schema \stackrel{def}{=} \lambda\, S, S' : Schema. \\
\quad [tmp \stackrel{def}{=} join\ S\ S'] \\
\quad [newsig \stackrel{def}{=} (\pi_1\, tmp)] \\
\quad [coercion1 \stackrel{def}{=} (\pi_2 tmp)] \\
\quad [coercion2 \stackrel{def}{=} coerce\ newsig\ S'.sig] \\
case \\
\quad (\lambda f : (Typify\ newsig) \rightarrow (Typify\ S'.sig). \\
\qquad (newsig, \lambda\, s : Typify\ newsig.\, (S.pred\ (coercion1\ s)) \wedge (S'.pred\ (f\ s)))) \\
\quad (\lambda\, x : unit.\, Absurd_schema) \\
\quad coercion2 \\
: Schema \rightarrow Schema \rightarrow Schema
\end{array}$$

With this method we can define all the operations of the Schema Calculus. However we need to compute lots of coercions and reasoning about these turns out to be difficult. For instance, *coerce* makes use of a simple function called *coerce_Ztypes* which takes two *Ztype*s z and z' and attempts to find mappings to and from $Typ\ z$ and $Typ\ z'$. This function has the type:

$$\Pi\, z, z' : Ztype.\, (((Typ\ z) \rightarrow (Typ\ z')) \times ((Typ\ z') \rightarrow (Typ\ z))) + unit$$

This function is defined by induction on *Ztype*s in a straightforward way. It succeeds when z and z' are identical, in which case it returns functions that are extensionally equal to the identity function on the appropriate type. So we tried to prove the following result:

$$\begin{array}{l}
\forall\, z : Ztype.\, \exists f, f' : (Typ\ z) \rightarrow (Typ\ z). \\
\qquad (coerce_Ztypes\ z\ z = in1\ (f, f')) \wedge (\forall\, x : Typ\ z.\, (f\ x = x) \wedge (f'\ x = x))
\end{array}$$

However we found that in order to prove it we needed to assume extensionality for functions. The reason for this has to do with way that *coerce_Ztype* is defined on *Ztypes* of the form $funT\ z\ z'$. The coercions that are found in this case are of type $((Typ\ z) \rightarrow (Typ\ z')) \rightarrow ((Typ\ z) \rightarrow (Typ\ z'))$ They map a function f to another function created by pre- and post-composing f with coercions between z and z and between z' and z'. This function is only extensionally equal to f.

Extensionality is not part of UTT and we would prefer not to have to assume it. Perhaps we can avoid it if we find a way of avoiding having to find coercions. We needed coercions because schema predicates were defined over specific signatures and therefore needed to be composed with coercions before they could be applied to other signatures. So what if we allow our predicates to be defined over all signatures? This leads us to explore a third method of representing Z schemas.

Method 3: Syntactic names in programs The idea here is to introduce a new type of programs, over which schema predicates will be defined. Whereas before we used UTT's typing rules to determine whether a program matched a signature, we will now have to explicitly define a matching relation between programs and signatures.

We continue to use the types *Ident*, *Ztype* and *Signature* and the semantic function *Typ*. However instead of representing programs as tuples, we use a more syntactic representation in which values are associated with syntactic names and types. We call these programs *Structures* since they are reminiscent of Structures in the programming language SML.

Definition 4 Structure.

$$Structure \stackrel{def}{=} list\ (\Sigma\, p : Ident \times Ztype.\ Typ\ (\pi_2 p)) : Type$$

We have to write some functions to handle these syntactic programs. First we define a function which simply extracts a *Signature* from a *Structure*:

Definition 5 extract_sig.

$$\begin{array}{l} extract\_sig \stackrel{def}{=} map\ \ \lambda\, x : str\_item.\ \pi_1 x \\ : Structure \rightarrow Signature \end{array}$$

The next function, *matches*, checks whether the names and identifiers in a *Structure* are exactly the same as those in a given *Signature*.

Definition 6 matches.

$$\begin{array}{l} matches \stackrel{def}{=} \lambda\, S : Signature.\ \lambda\, str : Structure.\ S \stackrel{bool}{=} (extract\_sig\ str) \\ : Signature \rightarrow Structure \rightarrow bool \end{array}$$

We redefine *lookup* to work with our new representation of programs.

Definition 7 lookup.

$$\begin{aligned}
&lookup \stackrel{def}{=} \lambda\, x : sig_item.\\
&\quad list_iter\ (in1\ void)\\
&\qquad\qquad (\lambda\, y : str_item.\ \lambda\, prev : (unit + str_item).\\
&\qquad\qquad\quad if\ (x \stackrel{bool}{=} (\pi_1 y))\ (in2\ y)\ prev)\\
&: sig_item \rightarrow Structure \rightarrow (unit + str_item)
\end{aligned}$$

The next function *restrict*, attempts to cut down a structure so that it has only those components specified in a given *Signature*; *restrict* fails if the *Signature* requires components that are not in the *Structure*.

Definition 8 restrict.

$$\begin{aligned}
&restrict \stackrel{def}{=} \lambda\, str : Structure.\\
&\quad list_iter\ (in2\ nil_str)\\
&\qquad\qquad (\lambda\, x : sig_item.\ \lambda\, prev : (unit + Structure).\\
&\qquad\qquad\quad case\ \lambda_ : unit.\ in1\ void\\
&\qquad\qquad\qquad \lambda\, s : Structure.\ case\ \ \lambda_ : unit.\ in1\ void\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \lambda\, si : str_item.\ in2\ (cons\ si\ s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (lookup\ x\ str)\\
&\qquad\qquad\qquad prev)\\
&: Structure \rightarrow Signature \rightarrow (unit + Structure)
\end{aligned}$$

An example of a syntactic program is the following:

$$prog \stackrel{def}{=} [(\text{‘}x\text{’}, natT, 0), (\text{‘}b\text{’}, boolT, true)] : Structure$$

If we define *sig* to be the signature $[(\text{‘}b\text{’}, boolT)]$ then $(matches\ sig\ prog)$ evaluates to *false* and $(restrict\ prog\ sig)$ evaluates to the *Structure* $[(\text{‘}b\text{’}, boolT, true)]$.

Our new definition of the type *Schema* is:

Definition 9 Schema.

$$Schema \stackrel{def}{=} Signature \times (Structure \rightarrow Prop) : Type$$

We redefine the projections *.sig* and *.pred* to use this new representation of schemas.

Schema predicates become more complicated since we have to explicitly allow for the possibility that *lookup* might fail. The predicates and relations that we use in writing schema predicates need to be redefined with this in mind. For instance in the definition of the schema S we replace $\leq : nat \rightarrow nat \rightarrow Prop$ with $\preceq : (unit + str\_item) \rightarrow (unit + nat) \rightarrow Prop$ which is defined as the proposition *absurd* in the case where either of its first arguments happens to be $in1\ void$. We also replace $length : \Pi\, t\!:\!Type.\,(list\ t) \rightarrow nat$ with $Length : (unit + str\_item) \rightarrow (unit + nat)$. Here is the new definition of this schema:

$$S\_sig \stackrel{def}{=} [(\text{`}x\text{'}, natT), (\text{`}y\text{'}, listT\ natT)]$$
$$S\_pred \stackrel{def}{=} \lambda\, str\!:\!Structure.$$
$$(lookup\ (\text{`}x\text{'},\ natT)\ str) \preceq Length\ (lookup\ (\text{`}y\text{'},\ (listT\ natT))\ str)$$
$$S \stackrel{def}{=} (S\_sig, S\_pred)$$

Another example is the new definition of $Absurd\_Schema$:

Definition 10.

$$Absurd\_schema \stackrel{def}{=} ([\,], \lambda\, s\!:\!Structure.\ absurd)\!:\!Schema$$

We need to place some conditions on schemas in order to exclude some badly behaved predicates. We would like predicates to remain true of structures if they are enriched by adding new components. Therefore we don't want to allow, for instance, a predicate that says that a certain identifier does not occur in a structure.

The first property we require of a schema says that any structure *str* which satisfies the schema's predicate must be capable of being restricted by the schema's signature, yielding a structure str' which must also satisfy the schema's predicate.

Definition 11.

$$\begin{array}{l} Condition1 \stackrel{def}{=} \lambda\, S\!:\!Schema. \\ \quad \forall\, str\!:\!Structure.\,(S.pred\ str) \Rightarrow case\ \begin{array}[t]{l} \lambda\,\_\!:\!unit.\ absurd \\ \lambda\, str'\!:\!Structure.\ S.pred\ str' \\ (restrict\ str\ S.sig) \end{array} \\ :Schema \rightarrow Prop \end{array}$$

The second condition is almost the converse of the first. This says that if a structure *str* can be restricted successfully by a schema's signature to give another structure which satisfies that schema's predicate, then *str* itself must satisfy the schema's predicate.

Definition 12.

$$\begin{array}{l} Condition2 \stackrel{def}{=} \lambda\, S\!:\!Schema. \\ \quad \forall\, str\!:\!Structure.\ case\ \begin{array}[t]{l} \lambda\,\_\!:\!unit.\ trueProp \\ \lambda\, str'\!:\!Structure.\,(S.pred\ str') \Rightarrow S.pred\ str \\ (restrict\ str\ S.sig) \end{array} \\ :Schema \rightarrow Prop \end{array}$$

A valid schema is one that satisfies both $Condition1$ and $Condition2$.

Definition 13 Valid_schema.

$$\begin{array}{l} Valid\_schema \stackrel{def}{=} \lambda\, S : Schema.\, (Condition1\ S) \wedge (Condition2\ S) \\ : Schema \rightarrow Prop \end{array}$$

The user will need to prove that this condition is satisfied whenever a new schema is defined. We have done this for the example schema S; the proof is lengthy but straightforward. We will also have to show that all our basic schemas and schema operations have or preserve this property.

Proposition 14 Absurd_schema_valid.

$$Valid\_schema\ Absurd\_schema$$

Proof. Trivial.

With these more syntactic programs, our definitions of schema operations become much simpler since there is no longer any need to compute or keep track of coercions. The function *join*, for instance, gets redefined in a simpler way. This definition makes use of a simple *bool*-valued function *member_ident* which checks whether a given identifier occurs in a given signature and is defined in an obvious way.

Definition 15 join.

$$\begin{array}{l} join \stackrel{def}{=} \lambda\, S, T : Signature. \\ \quad list\_iter\ T \\ \qquad\qquad \lambda\, x : sig\_item.\ \lambda\, prev : Signature. \\ \qquad\qquad\qquad if\ (member\_ident\ (\pi_1 n)\ prev)\ prev\ (cons\ x\ prev) \\ \qquad\qquad S \\ : Signature \rightarrow Signature \rightarrow Signature \end{array}$$

Here is the new definition of schema conjunction. We make use of a new function $consistent : Signature \rightarrow Signature \rightarrow bool$ which checks whether two signatures are consistent. We omit the definition of this function because it is long. Its behaviour is partially characterised by Lemma 33 in Appendix A.

Definition 16 And_schema.

$$\begin{array}{l} And\_schema \stackrel{def}{=} \lambda\, S, S' : Schema. \\ \qquad if\ (consistent\ S.sig\ S'.sig) \\ \qquad\quad ((join\ S.sig\ S'.sig),\ \lambda\, s : Structure.\, (S.pred\ s) \wedge (S'.pred\ s)) \\ \qquad\quad Absurd\_schema \\ : Schema \rightarrow Schema \rightarrow Schema \end{array}$$

We show that schema conjunction preserves the property *Valid_schema*. The lemmas referred to in this proof are given in Appendix A.

Proposition 17 And_preserves_validity.

$$\begin{array}{l}\forall\, S, T : Schema.\\ \quad (unique\_idents\ S.sig) \Rightarrow (unique\_idents\ T.sig) \Rightarrow\\ \quad\quad (Valid\_schema\ S) \Rightarrow (Valid\_schema\ T) \Rightarrow\\ \quad\quad\quad Valid\_schema\ (And\_schema\ S\ T)\end{array}$$

Proof. We introduce all the hypotheses and then consider the two possible values of $(consistent\ \ S.sig\ \ T.sig)$. In the case where this is equal to *false*, $And\_schema\ S\ T$ reduces to $Absurd\_schema$ for which we use Proposition 14.

In the case where S and T are consistent our goal is reduced to finding a proof of the following. Here the ? is the name of an existential variable, standing for the proof that we have to construct.

$$\begin{array}{l}? : Valid\_schema\\ \quad\quad ((join\ S.sig\ \ T.sig), \lambda\, str : Structure.\, (S.pred\ str) \wedge (T.pred\ str))\end{array}$$

We expand the definition of $Valid\_schema$ and then do an and-introduction which gives us two subgoals. The first of these is the following:

$$\begin{array}{l}?_{28} : \forall\, str : Structure.\, ((S.pred\ str) \wedge (T.pred\ str)) \Rightarrow\\ \quad\quad (case\ \lambda\, \_ : unit.\ absurd\\ \quad\quad\quad \lambda\, str' : Structure.\, ((S.pred\ str') \wedge (T.pred\ str'))\\ \quad\quad\quad (restrict\ str\ (join\ S.sig\ \ T.sig)))\end{array}$$

In other words we are being asked to show that if a structure str satisfies the predicates of both S and T, then it can be successfully restricted by the signature $(join\ \ S.sig\ \ T.sig)$ to give a new structure str' which also satisfies $S.pred$ and $T.pred$.

First we introduce into the proof context a structure str and a hypothesis H_2 stating that str satisfies $S.pred$ and $T.pred$. From H_2 and from the assumptions that S and T are valid schemas we can conclude that str can be successfully restricted by each of the signatures $S.sig$ and $T.sig$. We can then use Lemma 29 to derive that str can be successfully restricted by $(join\ S.sig\ \ T.sig)$, giving a new structure str'.

Now we have to show that str' satisfies $S.pred$ and $T.pred$. We use Lemma 30 to show that $(restrict\ str'\ S)$ is equal to $(restrict\ str\ S)$. Since S is a valid schema, and str satisfies $S.pred$, we can conclude that str' must also satisfy $S.pred$. We show that str' also satisfies $T.pred$ in a similar way, using instead Lemma 31 to show that $(restrict\ str'\ T)$ equals $(restrict\ str\ T)$. This concludes the proof of the first subgoal.

The second subgoal is almost the converse of the first:

$$\begin{array}{l}?_{29} : \forall\, str : Structure.\\ \quad\quad (case\ \lambda\, \_ : unit.\ trueProp\\ \quad\quad\quad \lambda\, str' : Structure.\, ((S.pred\ str') \wedge (T.pred\ str')) \Rightarrow\\ \quad\quad\quad\quad\quad\quad\quad\quad ((S.pred\ str) \wedge (T.pred\ str))\\ \quad\quad\quad (restrict\ str\ (join\ S.sig\ \ T.sig)))\end{array}$$

Here we have to show that if a structure *str* can be successfully restricted to the signature (*join* $S.sig$ $T.sig$) giving a new structure str', such that str' satisfies the predicates of S and T, then *str* must satisfy these predicates as well.

We introduce a structure *str*. Next we use a LEGO library lemma $in1\_or\_in2$ to examine the two possible results of restricting *str* to (*join* $S.sig$ $T.sig$). In the case where the result is ($in1$ *void*), our goal is reduced to proving *trueProp*, which is trivially true.

The remaining case is where *str* can be restricted successfully, giving a new structure str'. In this case we are reduced to having to show:

$$?_{242}:((S.pred\ str') \wedge (T.pred\ str')) \Rightarrow ((S.pred\ str) \wedge (T.pred\ str))$$

We use Lemmas 43 and 44 to show that *str* can be successfully restricted by $S.sig$ and $T.sig$, respectively. We use Lemma 30 to show that (*restrict* str' $S.sig$) equals (*restrict* *str* $S.sig$), and we use Lemma 31 to show the corresponding result for $T.sig$. With these in our proof context, we can then use the assumptions that S and T are valid to complete the proof. □

An implementation of a schema S is a structure *str* paired with a proof of (*Implements* S *str*) where *Implements* is defined as

Definition 18 Implements.

$$\begin{aligned}&Implements \stackrel{def}{=} \lambda S:Schema.\ \lambda str:structure.\\ &\qquad\qquad (matches\ S.sig\ str = true) \wedge (S.pred\ str)\\ &:Schema \rightarrow Structure \rightarrow Prop\end{aligned}$$

This states that an implementation of a schema is a *Structure* which exactly matches the schema's signature and which satisfies the schema's predicate. It may turn out that requiring an exact match with the signature is too restrictive, but for now this is the definition that we use.

We have succeeded in proving introduction and elimination rules for the operation *And_schema*. They hold for valid schemas under the reasonable condition that identifiers can appear at most once in any of the signatures and structures involved. This condition is expressed by the following predicate.

Definition 19 unique_idents.

$$\begin{aligned}&unique\_idents \stackrel{def}{=}\\ &\quad list\_rec\ trueProp\\ &\qquad\quad \lambda x:sig\_item.\ \lambda l:Signature.\ \lambda prev:Prop.\\ &\qquad\qquad\qquad ((member\ x\ l) = false) \wedge prev\\ &:Signature \rightarrow Prop\end{aligned}$$

These results make use of a predicate named *restricts_to_impl*, defined below, which says two things: that a given *Structure* can be successfully restricted to the signature of a given *Schema*, and that the new *Structure* obtained by this *restrict*ion is an implementation of the given *Schema*.

Definition 20 restricts_to_impl.

$$
\begin{array}{l}
\textit{restricts\_to\_impl} \stackrel{def}{=} \lambda\, S : \textit{Schema}.\ \lambda\, \textit{str} : \textit{Structure}. \\
\quad\quad \textit{case}\ \lambda\, \_ : \textit{unit}.\ \textit{absurd} \\
\quad\quad\quad\quad \lambda\, \textit{str}' : \textit{Structure}.\ \textit{Implements}\ S\ \textit{str}' \\
\quad\quad\quad\quad (\textit{restrict str}\ S.\textit{sig}) \\
: \textit{Schema} \rightarrow \textit{Structure} \rightarrow \textit{Prop}
\end{array}
$$

The following result gives a kind of introduction rule for schema conjunction. If we are given a *Structure* *str* which can be restricted to give implementations of two *Schemas* S and S', then we can conclude from this theorem that *str* can be restricted to an implementation of *And_schema* S S'.

Proposition 21 And_schema_intro.

$$
\begin{array}{l}
\forall\, S, T : \textit{Schema}.\ \forall\, \textit{str} : \textit{Structure}. \\
\quad\quad (\textit{Valid\_schema}\ S) \Rightarrow (\textit{Valid\_schema}\ T) \Rightarrow \\
\quad\quad (\textit{unique\_idents}\ S.\textit{sig}) \Rightarrow (\textit{unique\_idents}\ T.\textit{sig}) \Rightarrow \\
\quad\quad (\textit{unique\_idents}\ (\textit{extract\_sig str})) \Rightarrow \\
\quad\quad\quad (\textit{restricts\_to\_impl}\ S\ \textit{str}) \Rightarrow \\
\quad\quad\quad (\textit{restricts\_to\_impl}\ T\ \textit{str}) \Rightarrow \\
\quad\quad\quad \textit{restricts\_to\_impl}\ (\textit{And\_schema}\ S\ T)\ \textit{str}
\end{array}
$$

Proof. We begin by case analyses on the results of restricting *str* to $S.sig$ and to $T.sig$. In the cases where either of these restrictions fails we can easily obtain a contradiction with the appropriate hypothesis. For instance, if we have that ($\textit{restrict str}\ S.\textit{sig}$) is equal to $in1\ void$, we can use this to reduce the hypothesis ($\textit{restricts\_to\_impl}\ S\ \textit{str}$) to a proof of *absurd*.

So the interesting case is where *str* can be successfully restricted to both $S.sig$ and $T.sig$, yielding new structures *strS* and *strT*, respectively. Here is part of what our current proof context looks like:

$$
\begin{array}{l}
H_1 : \textit{restrict str}\ S.\textit{sig} = \textit{in}2\ \textit{strS} \\
H_4 : \textit{restrict str}\ T.\textit{sig} = \textit{in}2\ \textit{strT} \\
H_2 : \textit{Implements}\ S\ \textit{strS} \\
H_5 : \textit{Implements}\ T\ \textit{strT}
\end{array}
$$

Lemma 29 tells us that *str* can be successfully restricted to the signature ($\textit{join}\ S\ T$), yielding a new structure t. We add these facts to our proof context.

Eventually we wish to show that this structure t is an implementation of the schema *And_schema* S T. As a first step towards proving this we make the following two claims:

$$
\begin{array}{l}
?88 : S.\textit{pred str} \\
?89 : T.\textit{pred str}
\end{array}
$$

By applying Lemma 37, *join_is_consistent*, to these two claims, as well as to the appropriate hypotheses, we obtain a proof that ($\textit{consistent}\ S.\textit{sig}\ T.\textit{sig}$)

equals *true*. This allows us to conclude that the signature of $(And\_Schema\ S\ T)$ is equal to $(join\ S.sig\ T.sig)$. Since we know that t is the result of restricting *str* to this signature, our goal can be reduced to the following two subgoals:

$$?118 : is\_true\ (matches\ (join\ S.sig\ T.sig)\ t)$$
$$?119 : (S.pred\ t) \wedge (T.pred\ t)$$

We prove the first of these by using Lemma 39. Before we prove the second we make use of Proposition 17 to construct a proof that *And_schema S T* has the property *Valid_schema*. We use this proof, together with the fact that t is obtained by restricting *str* to the signature of $(And\_schema\ S\ T)$, and together with our two claims, to prove the second subgoal.

Next we have to prove the claims. These follow easily from the facts that schemas S and T have the property *Valid_schema*, and that *str* can be successfully restricted to give implementations of these schemas. □

The next two results can be thought of as left and right elimination rules for schema conjunction.

Proposition 22 And_schema_elim_l.

$$\begin{array}{l} \forall\, S, T : Schema.\ \forall\, str : Structure. \\ \quad (Valid\_schema\ S) \Rightarrow \\ \quad (unique\_idents\ S.sig) \Rightarrow (unique\_idents\ T.sig) \Rightarrow \\ \quad (unique\_idents\ (extract\_sig\ str)) \Rightarrow \\ \quad (is\_true\ (consistent\ S.sig\ T.sig)) \Rightarrow \\ \quad\ (restricts\_to\_impl\ (And\_schema\ S\ T)\ str) \Rightarrow restricts\_to\_impl\ S\ str \end{array}$$

Proof. By using LEGO's introduction tactics and rewriting with the hypothesis stating that $S.sig$ and $T.sig$ are consistent we arrive at a context including:

$$\begin{array}{l} str : Structure \\ H : Implements\ ((join\ S.sig\ T.sig), \lambda\, s : Structure.\, (S.pred\ s) \wedge (T.pred\ s)) \\ \qquad\qquad\qquad str \end{array}$$

Our goal at this point is:

$$? : restricts\_to\_impl\ S\ str$$

Next we make a claim, $?_{25}$ that *str* can be successfully restricted by $S.sig$, yielding a structure str'. We have

$$H_1 : restrict\ str\ S.sig = in2\ str'$$

We rewrite with H_1 and reduce our goal to having to show that str' is an implementation of S, that is, that it matches $S.sig$ and satisfies $S.pred$. We prove the first of these by applying Lemma 39. The second follows from H and the assumption that S has the property *Valid_schema*.

All that remains is to prove the claim, $?_{25}$. We prove this by Lemma 43, which then asks us to show that *str* can be successfully restricted by the signature $(join\ S.sig\ T.sig)$. This follows from Lemma 45 and the fact that by H, *str* matches $(join\ S.sig\ T.sig)$. □

Proposition 23 And_schema_elim_r.

$$
\begin{array}{l}
\forall\, S, T : Schema.\ \forall\, str : Structure. \\
\quad (Valid\_schema\ T) \Rightarrow \\
\quad (unique\_idents\ S.sig) \Rightarrow (unique\_idents\ T.sig) \Rightarrow \\
\quad (unique\_idents\ (extract\_sig\ str)) \Rightarrow \\
\quad (is\_true\ (consistent\ S.sig\ T.sig)) \Rightarrow \\
\quad\ \ (restricts\_to\_impl\ (And\_schema\ S\ T)\ str) \Rightarrow restricts\_to\_impl\ T\ str
\end{array}
$$

Proof. We prove this the same way we prove Proposition 22, using Lemma 44 instead of Lemma 43. □

4 Schema-level reasoning: an example

A technique that is commonly used in writing Z specifications is to specify a system by first writing a schema to describe the correct, or normal operation of the system, negating this schema to describe error conditions, and then disjoining these two schemas to specify a total operation. To support this pattern of specification building, we would like to prove, for instance, that the final schema produced by this process is, in some sense, total. Our encoding of the Schema Calculus allows us to prove results like this generally, for all schemas.

First we must define the operation of negating a schema. The negation of a schema S is a new schema whose signature is the same as that of S and whose predicate is true of all structures which can be successfully restricted to the signature of S and which satisfy the negation of the predicate of S.

Definition 24 Not_schema.

$$
\begin{array}{l}
Not\_schema \stackrel{def}{=} \lambda\, S : Schema. \\
(S.sig, \lambda\, str : Structure.\ case\ (\lambda\, \_ : unit.\ absurd) \\
\qquad\qquad\qquad\qquad\qquad\quad (\lambda\, str' : Structure. \neg(S.pred\ str')) \\
\qquad\qquad\qquad\qquad\qquad\quad (restrict\ str\ S.sig)) \\
: Schema \rightarrow Schema
\end{array}
$$

We also need the operation of schema disjunction. When we combine two schemas by this operation we obtain a new schema whose signature is the *join* of the two original signatures, and whose predicate is true of exactly those structures which can be restricted successfully to the new signature (and therefore to both of the original signatures) and which satisfy one or other of the original predicates.

Definition 25 Or_schema.

$$
\begin{array}{l}
Or\_schema \stackrel{def}{=} \lambda\, S, S' : Schema. \\
if\ (consistent\ S.sig\ S'.sig) \\
\quad ((join\ S.sig\ S'.sig), \\
\quad\ \lambda\, str : Structure.\ case\ (\lambda\, \_ : unit.\ absurd) \\
\qquad\qquad\qquad\qquad\qquad (\lambda\, str' : Structure.\, (S.pred\ str') \vee (S'.pred\ str')) \\
\qquad\qquad\qquad\qquad\qquad (restrict\ str\ (join\ (\pi_1 S)\ (\pi_1 S')))) \\
\quad Absurd\_schema \\
: Schema \rightarrow Schema \rightarrow Schema
\end{array}
$$

The "totality" theorem we will prove is the intuitionistic counterpart of the law of the excluded middle, expressed partly at the level of schemas. There are many variations on this statement that we can also prove, e.g. replacing the double negation by a double application of *Not_schema*, but this one seems likely to be the most useful.

Proposition 26 Totality.

$$
\begin{array}{l}
\forall\, S : Schema.\ \forall\, str : Structure. \\
\quad (unique\_idents\ S.sig) \Rightarrow (is\_in2\ (restrict\ str\ S.sig)) \Rightarrow \\
\qquad \neg\neg(restricts\_to\_impl\ (Or\_schema\ S\ (Not\_schema\ S))\ str)
\end{array}
$$

Proof. Unsurprisingly, this proof has the same basic structure as a proof of the analogous proposition: $\forall\, P : Prop.\, \neg\neg(P \vee \neg P)$. At various points we need to show that the signature of $(Or\_schema\ S\ (Not\_schema\ S))$ is the same as that of S, for which we use some of our lemmas about the *join* operation.

5 Comparison with other work on Z

Adequacy and other semantic accounts of Z

As a reference during this work we have used [5] which contains a lucid, semi-formal presentation of Z (though it does not claim to give a full semantics.) Hayes describes the operational behaviour of the schema-forming operators as they act upon the syntax of schema signatures and predicates. We believe that we have followed this account correctly as far as the effects on signatures is concerned. For predicates, however, we have had to make explicit the environments, or programs, to which they refer, which are left implicit in Hayes. We have also had to explicitly deal with the conditions in which schema combining operators fail. These considerations make it difficult to know how to formulate the statement of adequacy for the formation of schema predicates with respect to Hayes' presentation. As for the provability of predicates, we know that since UTT provides an intuitionistic logic we will not be able to prove as many theorems as in the classical logic used by Z.

The standard semantic account of Z is [2], which is a simplified version of the semantics in [15]. These present a comprehensive denotational account of

Z. We believe that there is a close relationship between our notion of program (*Structure*) and the semantic representation of schemas in these accounts ("structures" in [15] and "sets of bindings" in [2]), and that our work may be viewed as an implementation of something closely related to these semantics. This would be an interesting possibility to explore. We conjecture that there is some restricted version of the Z of [2] (minus schemas as types; with a simpler type system; using intuitionistic logic instead of classical; using only computable functions) which is adequately represented by my encoding. More work needs to be done on explicating the relationship between our work and [2] before this can be stated formally.

Proof support for Z

Since the language we have encoded is not Z, we cannot claim to have provided proof support for Z. However our encoding provides proof support for specifications structured by means of the Schema Calculus and written in a simple core language based on type theory rather than set theory. Our encoding also provides a notion of implementation (via programs written in constructive type theory) and enables LEGO to be used to verify that programs are implementations of specifications. Furthermore, our encoding allows us to prove general properties about the Schema Calculus operators themselves which can then be re-used when verifying programs. It is a "deep" embedding of the Schema Calculus in UTT, as contrasted with "shallow" embeddings in which the Schema Calculus operators are implemented within a metalanguage which "macro expands" schema operators and supplies the expanded forms as input to the proof-checker. Shallow embeddings do not allow the structure of specifications to be exploited when doing proofs.

Other work aimed at providing proof support for Z include: the ProofPower system [14] which is based a shallow embedding of Z in a HOL-like theorem-prover with a user interface that closely resembles Z notation; a shallow embedding of Z in HOL carried out by Bowen and Gordon [1]; the work of Martin [12] who has encoded W, a logic for Z, in the metalogical framework 2OBJ; the Z/EVES project [13] which uses a theorem prover for ZF set theory; the Balzac project [4] about which we know very little.

6 Conclusions and Future Work

The techniques we describe in this paper are promising. We can extend what we have done to encode all the operations of the Schema Calculus and prove more results of the kind demonstrated. These give us a framework for reasoning about specifications at the schema level. We can also attempt to extend our core language by adding new *Ztypes*, or even use a completely different core language such as an encoding of set theory.

To test the usability of this encoding we can encode and prove some theorems about a moderate-sized specification. This will reveal whether we have made useful choices in defining, for instance, an implementation, and whether our theorems about schema operations are appropriate for doing structured reasoning about specifications.

A major task would be to find out the relationship between our type-theoretic semantics for Z schemas and other proposed semantics. However even if we cannot do this in a formal way, we can still use examples to show how our encoding allows us to mimic the use of the Z notation within the LEGO proof-checker.

Another possible line of development would be to explore possible definitions of *refinement* within our system and to use Lego to study the relationship between refinement and the Schema Calculus operations.

7 Acknowledgements

We have benefited from discussions with Stuart Anderson, Healfdene Goguen, Claire Jones and from comments made by two anonymous referees. This work was partly supported by the British Council and partly by the Esprit BRA "Types".

References

1. J. Bowen and M. Gordon. Z and HOL, 1994. submitted to the '94 Z User Meeting.
2. S. Brien, J. Nicholls, et al. Z Base Standard version 1.0. Technical Report ZIP/PRG/92/121, Oxford University Computing Laboratory, 1992.
3. Dowek, Felty, et al. The Coq proof assistant user's guide, version 5.8. Technical report, INRIA-Rocquencourt, February 1993.
4. W. Harwood. Proof rules for Balzac. Technical Report WTH/P7/001, Imperial Software Technology, Cambridge, 1991.
5. I. Hayes. *Specification Case Studies.* Prentice-Hall International, 1987.
6. C. Jones and S. Maharaj. The LEGO library. Technical Report (forthcoming), LFCS, University of Edinburgh, 1994.
7. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press (forthcoming).
8. Z. Luo. Program Specification and Data Refinement in Type Theory. Technical Report ECS-LFCS-91-131, LFCS, University of Edinburgh, 1991.
9. Z. Luo. A Unifying Theory of Dependent Types: the Schematic Approach. In *Logical Foundations of Computer Science—Tver '92.* Springer-Verlag, 1992.
10. Z. Luo and R. Pollack. LEGO Proof Development System: user's manual. Technical Report ECS-LFCS-92-211, LFCS, University of Edinburgh, 1992.
11. S. Maharaj. Implementing Z in LEGO. MSc thesis, University of Edinburgh, 1990.
12. A. Martin. Encoding W: A logic for Z in 2OBJ. In *FME '93: Industrial-Strength Formal Methods*, Lecture Notes in Computer Science. Springer-Verlag, 1993.
13. M. Saaltink. Z and EVES. Technical Report TR-91-5449-02, Odyssey Research Associates, Ottawa, Canada, 1991.
14. ProofPower server. Send email to ProofPower-server@win.icl.co.uk.
15. J.M. Spivey. *Understanding Z: a specification language and its formal semantics.* Cambridge University Press, 1988.

A More Proof Descriptions

Lemma 27 ***restrict_to_tail.***

$$\forall\, sig : Signature.\ \forall\, x : sig\_item.\ \forall\, str : Structure.$$
$$is\_in2\ (restrict\ str\ (cons\ x\ sig)) \Rightarrow is\_in2\ (restrict\ str\ sig)$$

Proof. We prove this easily by manipulating the definition of *restrict*. □

Lemma 28 ***join_cons_cases.***

$$\forall\, sig, sig' : Signature.\ \forall\, x : sig\_item.$$
$$(join\ (cons\ x\ sig)\ sig' = join\ sig\ sig') \vee$$
$$(join\ (cons\ x\ sig)\ sig' = cons\ x\ (join\ sig\ sig'))$$

Proof. This is an easy proof that proceeds by first expanding the definition of *join*, then examining all possible values of $(member\_ident\ (\pi_1 x)\ (join\ sig\ sig'))$, and observing that the goal is satisfied in every case. □

Lemma 29 ***restrict_join.***

$$\forall\, sig, sig' : Signature.\ \forall\, str : Structure.$$
$$(is\_in2\ (restrict\ str\ sig)) \Rightarrow (is\_in2\ (restrict\ str\ sig')) \Rightarrow$$
$$is\_in2\ (restrict\ str\ (join\ sig\ sig'))$$

Proof. The proof is by a straightforward list induction on sig. In the case where sig is nil, $(join\ sig\ sig')$ reduces to sig', so the proof is immediate. In the case where sig is of the form $(cons\ x\ l)$ by making use of the induction hypothesis we get a proof context that includes the following:

$$H_1 : is\_in2\ (restrict\ str\ (cons\ x\ l))$$
$$H_2 : is\_in2\ (restrict\ str\ (join\ l\ sig'))$$

Next we use Lemma 28 to do a case analysis on the result of joining $(cons\ x\ l)$ and sig'. In the case where $(join\ (cons\ x\ l)\ sig')$ equals $(join\ l\ sig')$ the hypothesis H_2 proves the subgoal. In the next case we have

$$H_3 : join\ (cons\ x\ l)\ sig' = (cons\ x\ (join\ l\ sig'))$$

and our goal is

$$? : is\_in2\ (restrict\ str\ (join\ (cons\ x\ l)\ sig'))$$

We do a case analysis on the result of looking up x in str. In the case where *lookup* fails we obtain a contradiction with H_1 by using the lemma $in2\_not\_in1$ from the sums library. In the case where *lookup* succeeds we first use Lemma 27 to get a proof that str can be successfully restricted to the signature $(join\ l\ sig')$. We complete the proof by rewriting with H_3, expanding the definition of *restrict* and then substituting in the results of looking up x in str and of restricting str to $(join\ l\ sig')$. □

Lemma 30 restrict_join_l.

$$\begin{array}{l}\forall\, sig, sig' : Signature.\ \forall\, str : Structure.\\ \quad (is\_true\ (consistent\ sig\ sig')) \Rightarrow\\ \quad (unique\_idents\ sig) \Rightarrow (unique\_idents\ sig') \Rightarrow\\ \quad (is\_in2\ (restrict\ str\ sig)) \Rightarrow (is\_in2\ (restrict\ str\ sig'))\ \Rightarrow\\ \qquad case\ (\lambda\, \_ : unit.\ absurd)\\ \qquad\quad (\lambda\, str' : Structure.\ restrict\ str'\ sig = restrict\ str\ sig)\\ \qquad\quad (restrict\ str\ (join\ sig\ sig'))\end{array}$$

Proof. Omitted.

Lemma 31 restrict_join_r.

$$\begin{array}{l}\forall\, sig, sig' : Signature.\ \forall\, str : Structure.\\ \quad (unique\_idents\ sig) \Rightarrow (unique\_idents\ sig') \Rightarrow\\ \quad (is\_in2\ (restrict\ str\ sig)) \Rightarrow (is\_in2\ (restrict\ str\ sig'))\ \Rightarrow\\ \qquad case\ (\lambda\, \_ : unit.\ absurd)\\ \qquad\quad (\lambda\, str' : Structure.\ restrict\ str'\ sig' = restrict\ str\ sig')\\ \qquad\quad (restrict\ str\ (join\ sig\ sig'))\end{array}$$

Proof. Omitted.

In the next lemma we use the predicate *type_consistent*. Two signatures have this property if any identifier which appears in both signatures is paired with the same type in each occurrence.

Definition 32 type_consistent.

$$\begin{array}{l}type\_consistent \stackrel{def}{=}\ \lambda\, S, T : Signature.\ \forall\, i : Ident.\ \forall\, z, z' : Ztype.\\ \quad (is\_true\ (member\ (i, z)\ S)) \Rightarrow (is\_true\ (member\ (i, z')\ T)) \Rightarrow z = z'\\ : Signature \rightarrow Signature \rightarrow Prop\end{array}$$

Lemma 33 consistent_is_type_consistent.

$$\begin{array}{l}\forall\, sig, sig' : Signature.\\ \quad (unique\_idents\ sig) \Rightarrow (unique\_idents\ sig') \Rightarrow\\ \qquad (type\_consistent\ sig\ sig') \Leftrightarrow (is\_true\ (consistent\ sig\ sig'))\end{array}$$

Proof. Omitted.

Lemma 34 member_restrict_implies_member_orig.

$$\begin{array}{l}\forall\, sig : Signature.\ \forall\, str : Structure.\ \forall\, x : sig\_item.\\ \quad case\ \ \lambda\, \_ : unit.\ trueProp\\ \qquad \lambda\, str' : Structure.\ (is\_true\ (member\ x\ (extract\_sig\ str'))) \Rightarrow\\ \qquad\qquad\qquad\qquad (is\_true\ (member\ x\ (extract\_sig\ str)))\\ \qquad (restrict\ str\ sig)\end{array}$$

Proof. Omitted.

Lemma 35 restrict_equals_sig.

$$\begin{array}{l} \forall\, sig : Signature.\ \forall\, str : Structure. \\ \quad case\ \lambda\, \_ : unit.\ trueProp \\ \qquad \lambda\, str' : Structure.\ sig = extract\_sig\ str' \\ \qquad (restrict\ str\ sig) \end{array}$$

Proof. Omitted.

We introduce a predicate *self_consistent* which we use in the next lemma.

Definition 36 self_consistent.

$$\begin{array}{l} self\_consistent \stackrel{def}{=} \lambda\, S : Signature.\ type\_consistent\ S\ S \\ : Signature \rightarrow Prop \end{array}$$

Lemma 37 join_is_consistent.

$$\begin{array}{l} \forall\, S, T : Schema.\ \forall\, str : Structure. \\ \quad (Valid\_schema\ S) \Rightarrow (Valid\_schema\ T) \Rightarrow \\ \quad (unique\_idents\ S.sig) \Rightarrow (unique\_idents\ T.sig) \Rightarrow \\ \quad (self\_consistent\ (extract\_sig\ str)) \Rightarrow \\ \quad (S.pred\ str) \Rightarrow (T.pred\ str) \Rightarrow \\ \qquad is\_true\ (consistent\ S.sig\ T.sig) \end{array}$$

Proof. We introduce S, T, str, and the various hypotheses so that we are left having to show that $S.sig$ and $T.sig$ are consistent. We refine by Lemma 33, *consistent_is_type_consistent*, which gives us a new goal:

$$\begin{array}{l} ? : \forall\, i : Ident.\ \forall\, z, z' : Ztype. \\ \quad (is\_true\ (member\ (i, z)\ S.sig)) \Rightarrow (is\_true\ (member\ (i, z')\ T.sig)) \Rightarrow \\ \qquad z = z' \end{array}$$

We introduce i, z, z', and two hypotheses. From the assumptions that S and T are valid schemas and that their predicates are satisfied by str, we can deduce that str can be successfully restricted by $S.sig$ and $T.sig$, yielding structures $strS$ and $strT$, respectively.

We now try to prove that z and z' are equal by using the assumption that the signature of str is self-consistent. We try to show that (i, z) and (i, z') are both in the signature of str.

We know that (i, z) is a member of $S.sig$. By Lemma 35, *restrict_equals_sig*, we know that the signature of $strS$ equals $S.sig$, so (i, z) is a member of the signature $(extract\_sig\ strS)$. Since $strS$ is obtained by restricting str, Lemma 34 can be used to show that (i, z) is a member of str. By a similar argument we also show that (i, z') is in str. □

Lemma 38 restrict_matches_sig.

$$\begin{array}{l} \forall\, sig : Signature.\ \forall\, str : Structure. \\ \quad is\_true\ (case\ \lambda\, \_ : unit.\ true \\ \qquad\qquad \lambda\, str' : Structure.\ matches\ sig\ str' \\ \qquad\qquad (restrict\ str\ sig)) \end{array}$$

Proof. Omitted.

Lemma 39 restriction_matches.

$$\begin{array}{l} \forall\, sig : Signature.\ \forall\, str : Structure. \\ \quad (is\_in2\ (restrict\ str\ sig)) \Rightarrow \\ \quad is\_true\ (case\ \lambda\, \_ : unit.\ false \\ \qquad\qquad \lambda\, str' : Structure.\ matches\ sig\ str' \\ \qquad\qquad (restrict\ str\ sig)) \end{array}$$

Proof. We introduce sig, str and a hypothesis H. Using H we introduce a new structure str' which is the result of restricting str to sig. We use this to reduce the remaining goal to showing that str' matches sig. This we then prove by Lemma 38. □

Lemma 40 member_join_l.

$$\begin{array}{l} \forall\, sig, sig' : Signature.\ \forall\, x : sig\_item. \\ \quad (is\_true\ (consistent\ sig\ sig')) \Rightarrow \\ \quad (unique\_idents\ S) \Rightarrow (unique\_idents\ sig') \Rightarrow \\ \quad (is\_true\ (member\ x\ sig)) \Rightarrow \\ \qquad is\_true\ (member\ x\ (join\ sig\ sig')) \end{array}$$

Proof. Omitted.

Lemma 41 restrict_join_back_l_lemma1.

$$\begin{array}{l} \forall\, sig : Signature.\ \forall\, str : Structure.\ \forall\, x : sig\_item. \\ \quad (is\_in2\ (lookup\ x\ str)) \Rightarrow \\ \quad (is\_in2\ (restrict\ str\ sig)) \Rightarrow \\ \qquad is\_in2\ (restrict\ str\ (cons\ x\ sig)) \end{array}$$

Proof. This follows easily from the definition of $restrict$. □

Lemma 42 restrict_join_back_l_lemma2.

$$\begin{array}{l} \forall\, sig : Signature.\ \forall\, str : Structure.\ \forall\, x : sig\_item. \\ \quad (is\_true\ (member\ x\ sig)) \Rightarrow \\ \quad (is\_in2\ (restrict\ str\ sig)) \Rightarrow \\ \qquad is\_in2\ (lookup\ x\ str) \end{array}$$

Proof. Omitted.

Lemma 43 *restrict_join_back_l.*

$$\begin{array}{l}\forall\, sig, sig' : Signature.\ \forall\, str : Structure.\\ \quad (is\_true\ (consistent\ sig\ sig')) \Rightarrow\\ \quad (unique\_idents\ sig) \Rightarrow (unique\_idents\ sig') \Rightarrow\\ \qquad (is\_in2\ (restrict\ str\ (join\ sig\ sig'))) \Rightarrow is\_in2\ (restrict\ str\ sig)\end{array}$$

Proof. The proof is by list induction on sig. The base case follows easily from the definition of $restrict$. In the case where sig is of the form $(cons\ x\ l)$ we introduce a structure str and several hypotheses, leaving the following proof context and goal:

$$\begin{array}{l}H : \forall\, sig' : Signature.\ \forall\, str' : Structure.\ \ldots \Rightarrow\\ \quad (is\_in2\ (restrict\ str'(join\ l\ sig'))) \Rightarrow is\_in2\ (restrict\ str'\ l)\\ H_1 : is\_in2\ (restrict\ str\ (join\ (cons\ x\ l)\ sig'))\\ ?_{15} : is\_in2\ (restrict\ str\ (cons\ x\ l))\end{array}$$

Refining by Lemma 41 leaves us with the job of proving

$$\begin{array}{l}?_{19} : is\_in2\ (restrict\ s\ l)\\ ?_{20} : is\_in2\ (lookup\ x\ str)\end{array}$$

To prove the first of these we use the induction hypothesis H, which then gives us the task of showing that str can be successfully restricted to $(join\ l\ sig')$. We prove this by using $join\_cons\_cases$ to give us the two possible values of $(join\ (cons\ x\ l)\ sig')$, and showing that the subgoal follows from H_1 in either case.

Next we have to prove that looking up x in str is successful. By Lemma 42 we can show this by showing that x is a member of the signature $(join\ (cons\ x\ l)\ sig')$ and that str can be successfully restricted by this signature. The first of these follows from Lemma 40 and the second is in our list of hypotheses. □

Lemma 44 *restrict_join_back_r.*

$$\begin{array}{l}\forall\, sig, sig' : Signature.\ \forall\, str : Structure.\\ \quad (is\_in2\ (restrict\ str\ (join\ sig\ sig'))) \Rightarrow is\_in2\ (restrict\ str\ sig)\end{array}$$

Proof. We prove this easily by induction on sig. The case where sig is nil is immediate. In the case where sig is of the form $(cons\ x\ l)$ we use $join\_cons\_cases$ to examine the two possible values of $(join\ (cons\ x\ l)\ sig')$ and show that the goal follows from the induction hypothesis in either case. □

Lemma 45 *restrict_matching_sig.*

$$\begin{array}{l}\forall\, sig : Signature.\ \forall\, str : Structure.\\ \quad (unique\_idents\ sig) \Rightarrow\\ \quad (is\_true\ (matches\ sig\ str)) \Rightarrow\\ \qquad is\_in2\ (restrict\ str\ sig)\end{array}$$

Proof. Omitted. We believe that this result should remain true if the condition $(unique\_idents\ S)$ is removed, but we haven't found a proof of this.

B LEGO library functions and lemmas

$trueProp : Prop \stackrel{def}{=} \forall\, P : Prop.\, P \Rightarrow P$
$absurd : Prop \stackrel{def}{=} \forall\, P : Prop.\, P$
$list\_rec$ This lets us define functions from lists over a type s to a type t by supplying a base case of type t and a step function of type $\Pi\, head : s.\, \Pi\, tail : list\ s.\, \Pi\, prev : t.\, t$.
$list\_iter$. This lets us define functions from lists over a type s to a type t by supplying a base case of type t and a step function of type $\Pi\, head : s.\, \Pi\, prev : t.\, t$.
$length : \Pi\, t \mid Type.\, (list\ t) \rightarrow nat$ The $\mid$ here means that t is an implicit argument.
$if : \Pi\, t : Type.\, \Pi\, b : bool.\, \Pi\, tcase, fcase : t.\, t$ This allows us to define functions on $bool$s by case analysis.
$is\_in1 : \Pi\, s, t \mid Type.\, \Pi\, x : (s + t).\, Prop$ This is defined by $is\_in1\ x \stackrel{def}{=} \exists\, y : s.\, x = (in1\ y)$. Similarly we define is_in2.
$in2\_not\_in1 : \forall\, s, t : Type.\, \forall\, x : s.\, \neg(is\_in2\ (in1\ x))$
$in1\_or\_in2 : \forall\, s, t : Type.\, \forall\, x : (z + t).\, (is\_in1\ x) \vee (is\_in2\ x)$

This article was processed using the LaTeX macro package with LLNCS style