
Electronic version of an article published as

Available from: [http://dx.doi.org/10.1142/S0218194096000041](http://dx.doi.org/10.1142/S0218194096000041)


This is the author’s version of the work, posted here with the permission of the publisher for your personal use. No further distribution is permitted. You may also be able to access the published version from your library. The definitive version is available at [http://www.worldscinet.com/](http://www.worldscinet.com/).
On the animation of "not executable" specifications by Prolog

Leon Sterling*  Paolo Ciancarini  Todd Turnidge†
Case Western Reserve Univ.  University of Bologna  Case Western Reserve Univ.
Cleveland, Ohio - USA  Bologna, Italy  Cleveland, Ohio - USA

Abstract

An impediment to the widespread use of formal methods for software development is the difficulty in dealing with specifications, namely using them consistently in the software process. One approach to easing the management of specifications and improving their impact in the software process is animation, allowing developers to "execute" formal specifications as prototypes. This paper illustrates how Prolog can serve a multifaceted role for animating and prototyping specifications - as a target language, as the compilation/translation language, and to facilitate the advantages of formal methods through help in building formal proofs of properties such as correctness. Further, there is an implicit claim that, because of the correctness and directness of the translation, a subset of Z, not definitively established here, can be viewed as equivalent to a subset of Prolog.

1 The need to animate specifications

Formal specification of requirements of software systems has been advocated as a way to improve software quality. Being informal can compromise software quality by admitting unsound or ambiguous choices during the specification phase. Formal specification allows for formal derivation of important properties of the desired system, thus their full inclusion in the software development process has been advocated [6].

Several popular formal specification languages have emerged as candidates for widespread use. The best known formal specification languages, such as VDM and Z, share the interesting property that their basic theory is broad and high-level. Consequently, they have no executable operational semantics [8]. So there is a problem: these languages are intended to make simple the proving of important properties of the software system specification being studied, but they cannot help in establishing

*Current affiliation: Dept. of Computer Science, Univ. of Melbourne, Parkville, Vic. 3052 Australia
†Current affiliation: Dept. of Computer Science, Univ. of Wisconsin, Madison, Wisconsin
a very important property from a practical point of view: executability of the specification [7]. Further, software developers wanting to experiment with intermediate specifications to gain feedback from the client are deprived from using the computer to test the current version.

An approach has been developed specifically for these languages to allow for interactive testing: animation of the specifications. To animate a formal specification document means to produce an executable prototype by (automatic) transformations which preserve correctness and other "interesting" properties [9]. In this way validation of requirements by letting the customer "play" with the prototype becomes possible. For instance, the customer can "query" the prototype obtained from the specification to see if the basic behavior is acceptable, or can check if some error situations are adequately handled. In fact, testing and debugging of the specification document becomes possible in a "traditional" sense.

Since $Z$ is based on first order logic (and set theory), a good candidate as target language for animation is Prolog. In fact, both researchers and practitioners have attempted to develop methods of translating $Z$ schemas to Prolog code [16, 14, 11, 10, 3, 4, 5, 20, 12, 19]. However, despite the development of interesting prototypes, this research has had little impact. One factor is that the Prolog code produced has often been very inefficient. We believe that this is due to inferior transformations rather than to limitations of the approach.

In fact, we believe the strong position that it is possible to view a subset of $Z$ as syntactic sugar for executable Prolog code, and that code is suitable for helping to interactively develop the specification. It is compatible with the view that (a considerable part of) $Z$ has executable semantics. Some work has been done on developing a correct executable semantics for $Z$: a good starting point is [1].

More specifically, our position is:

- that the semantics of a suitable subset of $Z$, that which our transformation works upon, is clear;
- that the transformation for the subset is almost straightforward, as we will show;
- that the subset has sufficient expressivity for many applications.

Each of these claims deserve considerable discussion. As the focus of this special issue is the application of logic programming to software engineering, we have chosen not to enter the philosophical discussion that substantiating the claims would require. Rather we choose to show by example how effective the transformation is.

The paper is organized as follows. The next section surveys the two major approaches to Prolog-based animation of $Z$ specifications. Section 3 sketches a compiler, tzc, for translating $Z$ schemas into efficient Prolog code, thereby demonstrating equivalence between a subset of $Z$ and a subset of Prolog. Section 4 sketches how a proof could be constructed to establish that the translation preserves correctness. The final section discusses the significance of the work and gives conclusions.
2 Prolog-based approaches to animation

Loosely, there are two approaches to viewing the animation of a $Z$ specification. The first focuses on the characteristic nature of a $Z$ specification based on typed sets. For example, as presented in [1] the $Z$ schema

- \[
\begin{array}{c}
\textit{Odd} \\
x : \mathbb{N} \\
x \mod 2 = 1
\end{array}
\]

could be animated by enumerating the elements of the type \textit{Odd}, namely the odd numbers. This is not the problem when building specifications. The odd numbers are the odd numbers, the primes the primes, permutations are permutations, and functions are functions. Animating should not rely on generating all possible values for objects included in the specification document.

A first systematic approach to $Z$ animation was developed by Knott and Krause in the SuZan project [11]. They studied the feasibility of $Z$ animation, producing a library of Prolog predicates implementing standard $Z$ constructs. They also developed a basic strategy for manual translation of $Z$ to Prolog. This strategy can be summarized by the following formula:

- \[
\text{Schema} \leftarrow \text{Schema.Signature} \land \text{Schema.Predicate}
\]

That means that possible values of variables in a schema are defined by the schema signature and constrained by the schema predicate. The approach they suggested is called \textit{generate and test} because the specification is translated to a Prolog program as follows. For each schema a predicate is built which generates all the possible states defined by the signature part of the schema. For instance, a signature declaration $S : \text{P} \text{E}$, that means $S$ has type "set of E", becomes in Prolog \texttt{powerset}(E,Es), \texttt{member}(S,Es). After translating the signature, the schema predicate becomes a goal which tests the value generated by the signature part. Clearly in this approach there is a problem of combinatorial explosion because typically the output variables can be instantiated to candidate solutions only by a complete enumeration of values of types in the schema signature. For instance, a schema like

- \[
\begin{array}{c}
\textit{Root} \\
\textit{value}? : 1..100 \\
\textit{root}! : 1..10
\end{array}
\]

is translated as follows:

\[
\text{root(Value,Root) :- } \\
\quad \text{enumeration_range}(1,100,\text{Type_of_Value}), \\
\quad \text{member(Value,\text{Type_of_Value}), } \% \text{ generate} \\
\quad \text{member(Root,[1,2,3,4,5,6,7,8,9,10]), } \% \text{ generate} \\
\quad \text{Value is Root } \_ \star \_ \text{Root } \_ \% \text{ test}
\]

3
The translation takes two steps. First, the signature is used to bind the declared variables according to their type. A library of Prolog predicates can be used for this: for instance, the predicate `enumeration_range(X,Y,Z)` used above generates a list `Z` containing all the integers in the interval `[X,Y]`. The second step consists of writing the predicates corresponding to the second half of the schema, which are used to test for solutions.

This approach can handle any specification document written in Z, but is completely impractical. Often no satisfactory answer will be obtained by a query, either because there are infinite answers, or because the generate and test loop will take too much time. Also, there is limited translation for the useful schema calculus, i.e., the algebra of schema operators which allow a concise and modular description of a combination of states or operations.

A way to overcome the combinatorial explosion is to avoid automatic generation, using instead direct data input in the queries to the animation [20].

Another generate-and-test approach, called EZ (Executing Z) has been suggested in [5]. It exploits Z modularity and schema calculus, however only a subset of Z can be translated. Primitive types allowed are only integers, sequences, and strings. Schema operators include `-`, `^`, `_` and `#1C`. However, using EZ the specifiers cannot use:

- `∃` and `∀`;
- axiomatic definitions;
- generic schemata;
- set and multiset operations;
- functions, relationships;
- schema operator `≫`;
- a few logical operators like `⇔` and `⇒`

The basic idea in EZ is to see the specification document as specifying a search through a solution space, where the space is made of the abstract states and the search operators are specification (delta) operations able to transform the states. The name of a schema becomes the name of a Prolog predicate; variables in the signature become variables of the Prolog predicates; Z predicates become constraints which go in the right part of the rule defining the Prolog predicate. The user adds to the predicates obtained as described above a number of control rules which apply operations to a state and obtain new states.

A different approach consists of transforming a specification document to an “executable” form, so that it is simple to obtain a Prolog program as illustrated above. For instance, in [4] the schemata are presented into a form such that translation to Prolog is immediate. The basic idea is to write “procedural” specifications directly in Z, i.e., the Z schemata are written in a form suitable to be immediately translated into procedures of the target language. This amounts to manipulating the specifications to obtain a prototype of the specification. Signatures are not translated in Prolog, and the goal is only to obtain a program that implements the specification.
Using generate and test we would need to test all possible combinations of values for output variables known' and birthday'. Instead, these variables are left uninstantiated and the predicates are immediately translated to a sequence of goals to be tried top-down.

In this way an operational semantics written in Prolog is given to the Z specification. The specifier must be aware of the declarative and procedural semantics of both Z and Prolog, because a poorly chosen order of predicates or an unconstrained I/O usage of variables can produce programs which loop forever. Moreover, not all Z predicates have a Prolog counterpart, for instance universally quantified statements about output variables or ranging on infinite domains have no translation.

The best way to show our prescribed method for viewing Z as Prolog is with an example. Our primary running example throughout the paper will be the well studied birthday book example given in great detail in Spivey’s tutorial on Z [13]. It will be used to demonstrate both the animation of specifications and the use of Prolog for translation of the specification from Z to Prolog.

Let us begin with the schema AddBirthday of the classic BirthdayBook example [13], at the same time reminding the reader of basic Z terminology:

```
BirthdayBook
known : P NAME
birthday : NAME "" DATE
known = dom birthday

AddBirthday
ΔBirthdayBook
name? : NAME
date? : DATE

name? \notin known
birthday' = birthday \cup \{ name? \mapsto date? \}
known' = dom birthday'
```

The two boxed sections are called schemas.

The first schema, BirthdayBook, describes the state of a system. In this case, it says that a BirthdayBook consists of two entities: known, which is some subset of elements of the type NAME, and birthday, which is some function from NAME to DATE. These two entities constitute the signature of the schema. The predicate below the dividing line defines the invariant for the schema: in this case, at all times, known has to be the domain of the function birthday. This schema is properly represented in Prolog as a compound data structure. Each component of the state would be an argument. A suitable choice of term is birthdaybook(known,birthday).

The second schema, AddBirthday, describes the operation of adding a person's birthday to our BirthdayBook. The line ΔBirthdayBook states that AddBirthday is going to change the state of our system. This line implicitly says that this schema
deals with four variables at least: *known* and *birthday*, the components of the state before the operation, and *known' and *birthday*', the components of the state after the operation is complete. The next two lines tell us that the operation takes two input values, *name* and *date*. These are the name of a person, and the date of that person’s birthday, respectively. The predicates below the dividing line describe how to add this name-date pair to the *birthday* function.

This schema is appropriately represented in Prolog by a predicate with two arguments, a term representing the state before the operation and a term representing the state after the operation. The predicate is defined by a clause whose body is the predicates defined in the schema.

A suitable Prolog implementation is:

```
birthdaybook(Known, Birthday ) :-
    domain(Birthday,T0),
    equal(Known,T0,subset(name)).
```

```
addbirthday(birthdaybook(Known,Birthday),
    birthdaybook(Known_1,Birthday_1),
    Name, Date
) :-
    not_element(Name,Known),
    arrow(Name,Date,T0),
    makeset([T0],T1),
    union(Birthday,T1,T2),
    equal(Birthday_1,T2,function(name,date)),
    birthdaybook(Known_1,Birthday_1).
```

The schema calculus is also very naturally represented in logic programming. A conjunction of schemas is represented by a conjunction of Prolog predicates with attention paid to correct sharing of variables. Disjunction goes to two rules, arguments get picked up automatically from state schemas.

For example, continuing from the birthday book example, the definition of RAddBirthday,

```
RAddBirthday \doteq (AddBirthday \land Success) \lor AlreadyKnown.
```

can be translated into two rules as follows:

```
raddbirthday(BBook,BBook1,Result) :-
    addbirthday(BBook,BBook1),
    success(Result).
```

```
raddbirthday(BBook,BBook1,Result) :-
    alreadyknown(BBook,Result).
```
Which arguments to choose comes straightforwardly from the schema definition.
Once the translation is done, the animation can be tested. Logical variables are named, and the results can be verified. An annotated sample run is given in the Appendix.

3 Writing a compiler from Z to Prolog

Prolog is a good language for writing compilers, translators, and interpreters. A survey of Prolog for parsing can be found in [2]. The original Prolog compiler was written in Prolog [18]. Warren's methods are adapted in Chapter 24 of [15] and used in tzc.

Tzc uses the standard compiler architecture including lexical analysis, syntax analysis, then compilation and assembly. Fig.1 shows such a structure.
To make the paper somewhat self-contained, in the next paragraphs we review the interesting features of tzc.

3.1 Lookup Tables and State Information

In Prolog it is not customary to store information, even state information, globally. Instead, any state information is usually kept in a logical variable, and passed in as an argument to any predicate that needs it. When the state needs to change, we need two arguments: the first, the state at the beginning of the operation, and the second, the state at the end of the operation. We advocate handling state in this way in the target program.

Our state information is stored in the form:

\[
\text{state(\text{Constants,Types,Schemas})+Number.}
\]
**Constants** is a dictionary of the constants encountered while translating, and their types. **Types** is a dictionary of the types we have encountered while translating, and their definitions. **Schemas** is a dictionary of the schemas we have encountered so far, along with a list of their arguments. **Number** is used in the creation of uniquely named temporary variables. Its value is the number that will be assigned to the next temporary variable generated.

In tzc, **Constants**, **Types**, and **Schemas** are each examples of what is known as an incomplete type in Prolog. Specifically, they are incomplete lists [15]. This is a list data structure in which the end of the list is an unbound logic variable. This is useful, because information can be added to the list easily.

Because **Constants**, **Types**, and **Schemas** are all incomplete lists, any operation which only needs to do dictionary inserts/lookups needs only have one state argument passed in. If an argument needs to increment the temporary variable counter, however, it will need to have an input state argument and an output state argument. This inconsistency is not necessary, and, future versions of tzc would use input and output state arguments for all operations.

Lexical analysis is standard though we use our own symbols due to historical accident.

### 3.2 Parsing

The next stage after lexical analysis is parsing. During parsing, a syntax tree is built from the tokenized Z source. Actually, this stage builds a list of syntax trees, one for each schema in the original text. The Prolog execution model is well-suited to parsing and tree-building; in fact, a DCG (definite clause grammar) notation is supported by Sicstus Prolog, the Prolog system in which tzc is implemented.

The idea behind a DCG is the following. The DCG notation is very much like the traditional context-free grammar notation used to specify programming languages, with the exception that DCG’s can pass arguments, allowing syntax or parse trees to be built as they parse. DCG rules are, conceptually, almost Prolog rules. The only real change that needs to be made to make them run in Prolog is that an additional argument, the text to be parsed, has to be added to the rule. This argument is in the form of a difference list, another example of an incomplete data type. A difference list consists of two variables, which, for the time being, we can think of as marking the beginning and end of a substring to be parsed. (At this point, we should note that when difference list data structure is used at other places in this prototype, we use the notation C1/C2 to represent it.) For a more detailed treatment of difference lists and DCG rules, the reader is referred to [15].

As a top-level example of a parsing rule, consider one of the parsing rules that handles schemas:

\[
\text{p_schema_def} \text{(schema(Name,Args,Formulas),State)} \rightarrow \\
\text{p_identifier(Name),} \\
[\text{hateq, lbracket}],
\]

---

1The example uses word names for Z symbols: hateq is \(\equiv\), lbracket is \(\{\), etc.
This rule can be given a clear reading. In order to parse a schema definition, first, an identifier should be parsed, yielding a name in Name. Then the `hateq` token and the `lbracket` token should be consumed. Next, the schema arguments should be parsed, yielding their tree representation in Args. Finally, we parse the formulas in the body of the schema, which returns the syntax tree entry for the schema body in the variable Formulas. State is our state information. It needs to be passed in to p_schema_body so that constants can be recognized by using the constant lookup table. The first argument of p_schema_def itself, schema(Name,Args,Formulas), is the syntax tree entry corresponding to the entire schema. It is built from the results passed back by its component parsing goals.

As a second example, we will consider one of the rules used in parsing schema arguments of the form `date? : DATE`. The rule is:

```
p_schema_def(in(Name)+Type) -->
    p_identifier(Name),
    [huh, colon],
    p_type_def(Type).
```

This rule says that to parse an input schema argument, first parse an identifier, yielding Name. Then consume a question mark followed by a colon. Then parse some type definition, and store the result in Type. This rule builds a syntax tree entry of the form,

```
in(Name)+Type
```

This means we have an input argument named Name of type Type. The rule given here supports a simplification of Z that allows a list of declaration names followed by a colon followed by a (power-typed) expression. A declaration name can have decorations such as `?`, which is a syntactic convention to indicate that a variable is input.

Finally, we will give an example of one of the rules used to parse the formulas in the body of a Z schema. Equations of the form:

```
< Expression >=< Expression >
```

are parsed by the rule:

```
p_formula(equal(E1,E2),State) -->
    p_exp(E1,State),
    [eq],
    p_exp(E2,State).
```

This means that if we want to parse an equality formula, we parse an expression, yielding the syntax tree in E1. The State parameter is passed into p_exp to help with parsing constants. Then we consume the equals sign (eq). Finally, we parse the
second expression, yielding $E_2$. This rule returns the syntax tree $\text{equal}(E_1, E_2)$ to represent the formula.

All of the parsing rules work in this general way. The parsing process, although well-understood and certainly not very complicated, is essential because information about the definitions of types and declarations of constants is collected and inserted into lookup tables during this stage. The complete parsing rules are given in [17].

One final note before we move on to compiling. There is an important syntactic construct in $Z$ which we have not seen in our examples so far. It is the schema expression. So far, our treatment has covered schemas specified in horizontal form, such as,

\[ AddBirthday \equiv [\Delta BirthdayBook; name? : NAME; date? : DATE | name? \notin known; birthday' = birthday \cup \{name? \leftrightarrow date?\}] \]

A second way a schema can be specified is by a schema expression:

\[ RAddBirthday \equiv (AddBirthday \land Success) \lor AlreadyKnown. \]

This says that $RAddBirthday$ is satisfied whenever the $AddBirthday$ schema and the $Success$ schema are satisfied or whenever the $AlreadyKnown$ schema is satisfied. Note that this definition makes no mention of $RAddBirthday$'s signature; it is implicitly defined in terms of the signatures of the components of its defining schema expression.

The syntax tree for this structure is:

\[
\text{schema_exp}(\text{raddbirthday}, \\
\quad \text{sch_or}(\text{sch_and}(\text{addbirthday}, \text{success}), \text{alreadyknown}))
\]

Trees of this variety will be utilized in later examples.

### 3.3 Compiling

Once the data is in tree form, we can begin the true work of compiling. To do this, we traverse the list of syntax trees, schema by schema, compiling the syntax trees. As we compile each schema, we put its signature, its list of arguments and their types, into a lookup table, because the signatures may be needed during the compilation of later schemas. (The complete compiling rules are listed in [17].)

We do not compile directly to Prolog rules because we want to explicitly refer to the names of variables. This means that a variable that will eventually become $\text{Name}$ is represented as $\text{var(name)}$ at this stage. The only other real difference is that the code generated by this step still maintains some vestigial type information (which will be dropped during the final phase). When we need to explicitly refer to this 'almost Prolog' stage we will call it near-Prolog. One should note that if Prolog contained the capability of explicitly referring to a variable by name, then the compilation could be done directly to Prolog code.

We have seen two different ways for a schema to be specified, the horizontal form, and the schema expression form. These two forms need to be compiled in very different ways.
There are two stages in the translation of a schema specified in horizontal form: translating the schema’s arguments and translating the schema’s predicates. The argument translation is, for the most part, a straightforward correspondence. The following table gives a listing of Z arguments, and their translations into Prolog:

<table>
<thead>
<tr>
<th>Z argument</th>
<th>Prolog Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔBirthdayBook</td>
<td>birthdaybook(Known,Birthday), birthdaybook(Known_1,Birthday_1)</td>
</tr>
<tr>
<td>ΞBirthdayBook</td>
<td>birthdaybook(Known,Birthday)</td>
</tr>
<tr>
<td>BirthdayBook</td>
<td>birthdaybook(Known,Birthday)</td>
</tr>
<tr>
<td>name [?, !, or nothing]</td>
<td>Name</td>
</tr>
</tbody>
</table>

We have already seen this applied in the AddBirthday example. The actual new information is on the second and third lines of the chart. On the second line, we see the Ξ notation, which is used to indicate an operation with no state change. The bare notation on the third line is used when there is no input state, as is the case with an initialization schema.

In the cases of the Δ, Ξ and bare notations, code needs to be generated to ensure that state invariants hold. In the case of Δ and the bare notation, we insert a call to the state schema goal. In the case of the Ξ schema, which means that the state does not change—we simply insert code to unify the incoming state with the outgoing state.

The predicate used to translate individual arguments is trans_args:

trans_args(SI, ZArgs, PArgs, Varlist, C1/C2)

SI is an input state, used to access the schema lookup table. ZArgs is a list of Z arguments to be translated into Prolog arguments. PArgs is used to return the translated list of near-Prolog arguments. Varlist is a list of all variables introduced in PArgs. It is used in later translation to make sure that only variables that are defined in the current scope are accessed. C1/C2 is a difference list of near-Prolog goals that are generated to verify that state invariants are not violated.

We explain some of the argument translation rules to get a taste of their general composition. First, consider the rule which translates input arguments:

trans_args(SI, [in(X)+Type|Xs], [var(X)+Type|Ys], [var(X)+Type|Varlist], C1/C2)

This says that, if we want to translate an argument of the form in(X)+Type, the corresponding near-Prolog argument is var(X)+Type. var(X)+Type is the only new variable introduced, so it is prepended to our Varlist. No code is generated for an
input argument. \texttt{trans\_args} calls itself recursively to translate the remainder of the list.

Now, consider the rule that translates $\Delta$ arguments:

\begin{verbatim}
trans\_args(SI,
    [delta(X)|Xs],
    [Functor,PrimedFunctor|Ys],
    Varlist,
    C1/C3
) :-
    schema\_lookup(X, Parts, SI),
    make\_function(X, Parts, Functor),
    primed(Parts, PrimedParts),
    make\_function(X, PrimedParts, PrimedFunctor),
    C1 = [PrimedFunctor|C2],
    trans\_args(SI, Xs, Ys, RestVars, C2/C3),
    append(PrimedParts, RestVars, T),
    append(Parts, T, Varlist).
\end{verbatim}

Suppose $X$ is our familiar birthdaybook. We first look up \texttt{birthdaybook} in our list of already defined schemas. This puts \texttt{[var\(\text{known}\), var\(\text{birthday}\)]} into \texttt{Parts}. We use this to generate a primed version \texttt{[var\(\text{prime\(\text{known}\)}\),var\(\text{prime\(\text{birthday}\)}\)]}, and then we build the record structures \texttt{birthdaybook(var\(\text{known}\),var\(\text{birthday}\))} and \texttt{birthdaybook(var\(\text{prime\(\text{known}\)}\),var\(\text{prime\(\text{birthday}\)}\))} in \texttt{Functor} and \texttt{PrimedFunctor}, respectively. Since the record structure also serves as a goal, we add \texttt{PrimedFunctor} onto our code difference list, to ensure that the output state satisfies the invariants. The $\Delta$ notation introduces four new variables, so we add \texttt{Parts} and \texttt{PrimedParts} to our \texttt{Varlist}, to signify this.

After compiling a schema's arguments, we need to compile its predicates. The inherent recursive structure of our syntax tree leads us to build a recursive hierarchy of translation predicates. The quickest way to explain how this is done is by example. We will describe the translation process from the bottom-up, beginning with the \texttt{trans\_term} predicate:

\begin{verbatim}
trans\_term(SI,SO,Varlist,Exp,C1/C2,ResultVar,ResultType)
\end{verbatim}

$SI$ is an input state, $SO$ is an output state, $Varlist$ is a list of variable names defined in the current scope, $Exp$ is an expression to be translated to near-Prolog, $C1/C2$ is a difference list of near-Prolog goals that, when executed will compute $Exp$, $ResultVar$ is the name of the variable in which the result of $C1/C2$ is stored, and $ResultType$ is the type of the result stored in $ResultVar$.

Consider the following version of the \texttt{trans\_term} predicate, used to translate expressions of the form \texttt{domain(SubExp)}:

\begin{verbatim}
trans\_term(SI,
    SO,
    \end{verbatim}
First of all, we call `trans_term` recursively to generate code (C1/C2) and type information (Type1) for our sub-expression, SubExp. After this, we know that V contains the name of a variable which would hold the value of SubExp after evaluation of C1/C2. Now we look up the definition of the type Type1 to ensure that it is indeed a function (domain is only defined on functions) and to find the type, DType, of elements in the function’s domain. We will need a variable to hold our result when we are finished, and the `new_var` goal generates a unique, new variable name (stored in NewV) for us. The final goal,

\[
\text{C2} = \text{[domain(V,NewV)|C3]}
\]

append a goal to our difference list which, when evaluated, will, thanks to the predicate library, set the variable named by NewV to the domain of the value of the variable named by V. The type of the result is `subset(DType)`.

This recursive style of evaluation is used to translate all expressions, down to the base cases of simple variables and constants.

Next we look at the level of the Z predicate. The predicate used to translate Z predicates is `trans_formula`:

\[
\text{trans_formula(SI, SO, Varlist, Formula, C1/C2) : -}
\]

SI is our input state, SO is our output state, Varlist is a list of variables defined in the current scope, Formula is a Z predicate to be translated, and C1/C2 is the near-Prolog code that, when executed, will succeed or fail if and only if the corresponding Z predicate would have succeeded or failed with similar variable bindings.

Suppose we have a predicate of the form:

\[
< \text{Expression} > = < \text{Expression} >.
\]

We said earlier that this was built into a syntax tree of the form `equal(A,B)`. The following rule is used to compile syntax trees of this form into near-Prolog:

\[
\text{trans_formula(SI,}
\]

SO,

Varlist,

equal(A,B),

C1/C4

\) : -

\[
\text{trans_term(SI, S, Varlist, A, C1/C2, Result, AType),}
\]
trans_term(S, SO, Varlist, B, C2/C3, Result2, BType),
type_check(AType, BType, SO),
C3 = [equal(Result, Result2, AType)|C4].

As we saw above, the two \texttt{trans\_term} goals generate code and type information for the two sub-expressions A and B, storing the results in the variables named by \texttt{Result} and \texttt{Result2}, respectively. The \texttt{type\_check} goal ensures that both expressions are of the same type (because we can only compare two values for equality if they are of the same type). The final goal,

\[ C3 = [\text{equal(Result, Result2, AType)}] | \text{C4} \]

appends a goal to our difference list which, when evaluated, will only succeed if the variables named by \texttt{Result} and \texttt{Result2} hold equivalent values when treated as the type \texttt{AType}. In this manner, all Z predicates are translated.

We have yet to deal with the second form of Z schema specification, the schema expression. Schema expressions have a natural representation in Prolog. Each schema reference corresponds to a goal, each schema-and simply corresponds to a conjunction of two goals, and a schema-or corresponds to the creation of two separate rules\(^2\), one for each of the arguments. For example, the schema expression for RAAddBirthday,

\[ RAAddBirthday \equiv (AddBirthday \land Success) \lor AlreadyKnown. \]

gets eventually compiled into the following two rules:

\begin{verbatim}
raaddbirthday(birthdaybook(Known,Birthday),
    birthdaybook(Known_1,Birthday_1),
    Name,
    Date,
    Result
) :-
addbirthday(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),Name,Date),
success(Result).
raaddbirthday(birthdaybook(Known,Birthday),
    birthdaybook(Known_1,Birthday_1),
    Name,
    Date,
    Result
) :-
alreadyknown(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),Name,Result),
\end{verbatim}

This translation is straightforward, the only complication being the arguments. In the example above, the Z specification of \textit{RAAddBirthday} makes no mention of

\footnotetext[2]{This could be done with "\textasciitilde" in Prolog, but our Prolog programming style prefers two rules.}
RAddBirthday's arguments; we need to compose these from the arguments of the component schemas. This information is stored for each schema in a lookup table.

One of the predicates used in translating schema expressions is:

\[
\text{trans} \_\text{schema} \_\text{sub} \_\text{exp}(\text{State}, \text{SubExp}, \text{Args}, \text{Body})
\]

\text{State} is the input state information. \text{SubExp} is a schema-sub-expression to be translated, \text{Args} is the list of arguments of the translated schema, and \text{Body} is a list of goals in the body of the translated schema.

As an example, schema-and's have a syntax tree of the form:

\[
\text{sch} \_\text{and}(\text{SubExp1}, \text{SubExp2})
\]

Consider the version of \text{trans_schema_sub_exp} which compiles schema-ands,

\[
\text{trans} \_\text{schema} \_\text{sub} \_\text{exp}(\text{State}, \\
\quad \text{sch} \_\text{and}(\text{Sub}1, \text{Sub}2), \\
\quad \text{Args}, \\
\quad \text{Body} \\
\) :- \\
\quad \text{trans} \_\text{schema} \_\text{sub} \_\text{exp}(\text{State}, \text{Sub}1, \text{Args}1, \text{Body}1), \\
\quad \text{trans} \_\text{schema} \_\text{sub} \_\text{exp}(\text{State}, \text{Sub}2, \text{Args}2, \text{Body}2), \\
\quad \text{disjoint} \_\text{merge}(\text{Args}1, \text{Args}2, \text{Args}), \\
\quad \text{append}(\text{Body}1, \text{Body}2, \text{Body}).
\]

The two \text{trans_schema_sub_exp} goals recursively determine the arguments and body for the sub-expressions \text{Sub}1 and \text{Sub}2.

Once we have these, the \text{disjoint_merge} goal combines the lists of arguments from both sub-expressions with no duplicates. The \text{append} goal joins the lists of goals from each sub-expression to form our combined list of goals.

Schema-or is currently handled only in the simple form we have shown in earlier examples (one schema-or, at the top level of the schema-expression). This is because, if we were to allow arbitrarily complex combinations of schema-or's and schema-and's, then we would need to introduce a mechanism for generating Prolog predicates for every schema-or that appears in a sub-expression. This is certainly possible, but, for our purposes, it is not necessary.

### 3.4 Output

After compilation is finished, all that remains to be done is to output the compiled code, transforming it from our near-Prolog into Prolog code in the process. Essentially, this amounts to “pretty-printing.” This task is accomplished by the display predicates, which recursively traverse the near-Prolog terms, translating and displaying the results. The complete output code is available in [17].

One other task is performed during the output stage; we traverse our type definitions, and when appropriate, output \text{generic_type} directives, which look like this:
generic_type(name).

These rules provide information for the tzc predicate library, so it can accurately access equality of values of these types. It is also used when constants of these types are referenced. We will discuss these issues more in our discussion of the tzc predicate library.

3.5 Predicate library

Tzc contains a predicate library. The predicates in the library do all of the ‘real work’ of evaluating Z expressions and Z predicates. The library also serves to hide the details of the underlying representations in Prolog of entities such as sets, functions, etc.

The following predicates have been implemented. Clearly this is only a small subset of the operations defined in the standard Z mathematical toolkit. However, they are sufficient to cover both the birthdaybook and checkpointing scheme given in Spivey’s tutorial.

- domain
- arrow
- union
- func_call
- set_of
- get_constant
- element
- not_element
- equal

The library provides a default behavior for what we call generic types. If a given type (like NAME in the BirthdayBook example) is considered generic, the library just treats the type values as Prolog terms; equality is unification, the value of a constant is its name, etc. By removing a directive, generic_type for a given type from the generated code, the user can override this default behavior. If the user does override the default, he will need to define equality, constant lookups, and the like for that type explicitly, which would be useful, for example, if unification were not a valid method of determining equality. Consider the situation, for example, where an entire equivalence class of values were to be considered equal (maybe a Prolog implementation of fractions, where 1/2 is considered equivalent to 2/4). The default behavior works well for all of our examples.

In its current incarnation, the predicate library implements sets as lists, with duplicates being allowed but ignored. This makes the union operation fast, but the equality operation a bit slower on sets. Functions are represented as subsets of cross-products.
The implementation of the library predicates tend to follow the conventional Prolog idioms, with the possible exception of equal(A,B,Type). Because sometimes one or the other of A and B are unbound logic variables, we need to explicitly handle each possibility. The reason equal needs the third argument Type is that, for some types, such as sets, unification is not a valid means of determining equality.

4 Correctness

What does it mean for the translation to be correct? Our view is that the translation is correct iff:

(a certain collection of terms satisfies a Z schema) iff (the Prolog representation of those terms satisfies the Prolog rule corresponding to that Z schema).

This can be represented with a diagram as follows.

The semantics of a Z document should be either operationally defined to be amenable to some kind of observation via some notion of abstract execution, as in [1], or its translation in Prolog, when animated, should allow queries whose answer correspond to observations on the abstract operational semantics. The correctness condition is that the diagram commutes in the sense of category theory.

We believe that it is straightforward to build a proof of correctness of tzc by piecing together a collection of smaller proofs. We outline the approach.

As discussed in the previous section, a library of correct Prolog equivalents of Z operations is assumed. A proof of correctness is needed for each predicate in the library. The intuition is that the Prolog predicate behaves ‘the way it is expected’. As an example, if the goal domain(F,D) predicate was included in a translated clause, we would expect to be able to say, that “if F is a valid representation of a function or partial function, then D is the Prolog representation of that function’s domain”.

An example involving more computation is using the Prolog predicate member to correctly reflect the semantics of the Z symbol for membership. The translation routines traverse the parse tree and produce a corresponding Prolog rule.

So there is a natural inductive structure of the proof corresponding to inductively defined parse trees. The inductive structure becomes:

If a translation predicate translates a subtree corresponding to a term, we need to prove that the generated goal(s) in some sense produce(s) the corresponding Prolog term.

This can be done by formalizing an argument by inspection. If a translation predicate translates a subtree corresponding to a logical formula, we need to prove that that logical formula is satisfied precisely in the case that the translation (into
Prolog goals of that subtree is satisfied. If a translation predicate translates a subtree corresponding to a schema, then the generated predicate satisfies the correctness criterion.

To take a specific example, consider the clause which translates the Z term \( element(A,B) \) into the Prolog term \( element(ResultA,ResultB) \):

```prolog
trans_formula(SI,S0,Varlist,element(A,B),C1/C4) :-
  trans_term(SI,S,Varlist,A,C1/C2,ResultA,AType),
  trans_term(S,S0,Varlist,B,C2/C3,ResultB,BType),
  type_check(subset(AType),BType),
  C3 = [element(ResultA,ResultB)|C4].
```

We need to show that \( trans\_formula \) translates a subtree of the Z specification which represents the element formula correctly. So, assume that \( SI \) is a valid input state, \( Varlist \) is a list of variables in the current scope, and \( element(A,B) \) is a subtree of our Z parse tree. We will show that

\[
trans\_formula(SI+S0,Varlist,element(A,B),C1/C4)
\]

holds iff \( C1/C4 \) is a difference-list representation of Prolog goals which are satisfied iff \( A \) is an element of \( B \) AND \( S0 \) is the appropriate output state.

We know that \( A \) and \( B \) are tree representations of Z terms. By a lemma for \( trans\_term \),

\[
trans\_term(SI+S0,Varlist,A,C1/C2,ResultA,AType)
\]

is satisfied in the case that

1. \( ResultA \) is a valid Prolog representation of \( A \);
2. \( A \) is of type \( AType \);
3. \( C1/C2 \) is a difference-list representing the goals necessary to generate the Prolog term;
4. \( S \) is a valid intermediate state after translating \( A \).

Another application of the same lemma yields

\[
trans\_term(S+S0,Varlist,B,C2/C3,ResultB,BType)
\]

is satisfied in the case that

1. \( ResultB \) is a valid Prolog representation of \( B \);
2. \( B \) is of type \( BType \);
3. \( C2/C3 \) is a difference-list representing the goals necessary to generate the Prolog term;
4. \( S0 \) is a valid intermediate state after translating \( B \).
The type_check goal does some checking to make sure that the original Z goals are well-formed. It is a convenience for development of Z specifications and for debugging the code.

The last line of the predicate C3=C[element(ResultA,ResultB)|C4] constructs a difference-list representation for the Prolog goal element(ResultA,ResultB).

We know that ResultA and ResultB are valid Prolog representations of terms A and B. By the library proof for the predicate element we know that the above is satisfied precisely in the case that whenever ResultA and ResultB are valid Prolog representations of some term A of type T and some term B of type subset of T, then term A must be an element of term B.

Therefore, when the Prolog goals represented by C1/C4 are satisfied, we know that A is an element of B and S0 is our appropriate output state. A similar argument is required for every template for a formula subtree.

Piecing together all of the translation rules which constitute tzc would lead to a proof that the translation is correct.

5 Discussion and Conclusions

The compiler tzc is a successful prototype for animation of Z specifications. It is able to transform sizable examples into working, efficient, and readable Prolog code. Thus an opportunity is provided to inspect the behavior of a specification before it is finalized, to see if it will work as expected.

The ability to compile Z draws attention to the blurry line between what constitutes specifying and what constitutes coding. In our opinion, the Z specification is, in some sense, code, because it can be systematically made executable. This contrasts with the opinion of a Z purist who would insist that a Z specification only states a problem and does not implement its solution. In any case, we are led to the question, if Z can sometimes be viewed as a coding language, can Prolog be a specification language?

What would be necessary to extend the compiler to handle a wider spectrum of Z specifications? This is an interesting question, and a good source for possible extensions to tzc. First of all, the tzc predicate library only implements the subset of the Z mathematical toolkit that is utilized in the examples we have been interested in. In order to handle the full range of Z predicates and expressions, we would need to augment the tzc predicate library considerably, providing a predicate for each operator we wished to add. Parsing and compilation rules would have to be added for these new operators as well, but these new rules would follow the general pattern already established. There would be a few other changes necessary as well. Currently, for ease of implementation, all Z operators are assumed to be of equal precedence and associate from left to right. Since this is not the case in true Z, the grammar rules would need to be modified accordingly.

Z provides many different operators for defining function types. We have injective functions, surjective functions, total functions, partial functions, etc. It should be noted that the current implementation of tzc makes no effort to preserve these properties; all functions are represented as subsets of cross-products internally, and the
library routines do not attempt to verify that any of these properties are preserved. In fact, in Prolog, things like types tend to be defined implicitly: a type is whatever it is. We never need define the type explicitly. We will see this in our example run of BirthdayBook in the Appendix. This implicit style in Prolog would make it hard to tie down certain properties about functions; for example, whether a function is total or partial becomes more of a philosophical issue than a practical one.

Acknowledgements

Leon Sterling was supported by NSF grant CCR-9303484. The initial motivation for the approach to compilation of Z specifications into readable Prolog code embodied in tzc arose from discussions between the first author and Bohdan Durnota of Monash University, Caulfield campus in late 1992. Specifications produced by students from Bohdan’s formal methods class were hand translated into Prolog. Useful followup discussions were provided by the ProSE group at CWRU. Part of the final write up happened at Melbourne University. Paolo Ciancarini acknowledges a partial support from Italian MURST and CNR. Todd Turnidge did the implementation of tzc as part of a CWRU Masters project. Comments by the reviewers were helpful and have improved the paper.

References


**Appendix**

The Prolog code corresponding to the birthday book example is the following.

```prolog
birthdaybook(Known,
    Birthday
  ) :-
    domain(Birthday,T0),
    equal(Known,T0,subset(name)).
```
addbirthday(birthdaybook(Known,Birthday),
    birthdaybook(Known_1,Birthday_1),
    Name,
    Date
) :-
    not_element(Name,Known),
    arrow(Name,Date,T0),
    makeset([T0],T1),
    union(Birthday,T1,T2),
    equal(Birthday_1,T2,function(name,date)),
    birthdaybook(Known_1,Birthday_1).

findbirthday(birthdaybook(Known,Birthday),
    birthdaybook(Known_1,Birthday_1),
    Name,
    Date
) :-
    element(Name,Known),
    func_call(Birthday,Name,T0),
    equal(Date,T0,date),
    birthdaybook(Known,Birthday)=birthdaybook(Known_1,Birthday_1).

remind(birthdaybook(Known,Birthday),
    birthdaybook(Known_1,Birthday_1),
    Today,
    Cards
) :-
    set_of(N, [element(N,Known),func_call(Birthd ay,N,T0),equal(T0,Today,date)], T1),
    equal(Cards,T1,subset(name)),
    birthdaybook(Known,Birthday)=birthdaybook(Known_1,Birthday_1).

initbirthdaybook(birthdaybook(Known,Birthday)) :-
    get_constant(emptys et,T0),
    equal(Known,T0,subset(name)),
    birthdaybook(Known,Birthday).

success(Result) :-
    get_constant(ok,T0),
    equal(Result,T0,report).

alreadyknown(birthdaybook(Known,Birthday),
    birthdaybook(Known_1,Birthday_1),
element(Name,Known),
get_constant(already_known,T0),
equal(Result,T0,report),
birthdaybook(Known,Birthday)=birthdaybook(Known_1,Birthday_1).

raddbirthday(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),
Name, Date, Result)
) :-

addbirthday(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),Name,Date),
success(Result).

raddbirthday(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),
Name, Date, Result)
) :-

alreadyknown(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),
Name,Result).

notknown(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),
Name, Result)
) :-

not_element(Name,Known),
get_constant(not_known,T0),
equal(Result,T0,report),
birthdaybook(Known,Birthday)=birthdaybook(Known_1,Birthday_1).

rfindbirthday(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),
Name, Date, Result)
) :-

findbirthday(birthdaybook(Known,Birthday),
birthdaybook(Known_1,Birthday_1),
Name, Result)
) :-
birthdaybook(Known, Birthday),
Name, Date),
success(Result).

rfindbirthday(birthdaybook(Known, Birthday),
birthdaybook(Known, Birthday),
Name, Date, Result
) :-
notknown(birthdaybook(Known, Birthday),
birthdaybook(Known, Birthday),
Name, Result).

rrremind(birthdaybook(Known, Birthday),
birthdaybook(Known, Birthday),
Today, Cards, Result
) :-
remind(birthdaybook(Known, Birthday),
birthdaybook(Known, Birthday),
Today, Cards),
success(Result).

Here is a sample run of the Prolog code.

?-initbirthdaybook(B1),
raddbirthday(B1,B2,todd,6-9,R2),
raddbirthday(B2,B3,leon,5-17,R3),
raddbirthday(B3,B4,leon,1-1,R4),
raddbirthday(B4,B5,mork,6-9,R5),
rrremind(B5,B6,6-9,Cards,R6),
rrfindbirthday(B6,B7,leon,BD7,R7),
rrfindbirthday(B7,B8,ringo,BD8,R8).

B1 == birthdaybook([],[]),
B2 == birthdaybook([todd],[[todd,6-9]]),
B3 == birthdaybook([todd,leon],[[todd,6-9],[leon,5-17]]),
B4 == birthdaybook([todd,leon],[[todd,6-9],[leon,5-17]]),
B5 == birthdaybook([todd,leon,mork],[[todd,6-9],[leon,5-17],[mork,6-9]]),
B6 == birthdaybook([todd,leon,mork],[[todd,6-9],[leon,5-17],[mork,6-9]]),
B7 == birthdaybook([todd,leon,mork],[[todd,6-9],[leon,5-17],[mork,6-9]]),
B8 == birthdaybook([todd,leon,mork],[[todd,6-9],[leon,5-17],[mork,6-9]]),
BD7 == 5-17,
BD8 == _858,
Cards == [mork,todd],
R2 == ok,
R3 == ok,
R4 == already_known,
R5 == ok,
R6 == ok,
R7 == ok,
R8 == not_known

The results of this sample run are exactly as expected.

The call to initbirthdaybook initializes the BirthdayBook to the term birthdaybook([],[]). The first call to raddbirthday succeeds, adding todd to the birthday function with a birthday of 6-9. Because the operation is successful, R2 is set to ok. The next call to raddbirthday successfully adds leon’s birthday to our BirthdayBook. The next call to raddbirthday is unsuccessful because leon’s birthday is already in the BirthdayBook. We can see this by looking at the result value: R4 is already_known. The next call to raddbirthday successfully adds mork’s birthday to the BirthdayBook. The call to rremind with an argument of 6-9 generates the set of all people whose birthdays are on 6-9 and stores this set in the variable Cards. In this case, there are two people listed in Cards, mork and todd. We then try to find leon’s birthday. Since this is in the BirthdayBook, BD7 is bound to 5-17 and R7 is set to success. The final call to rfindbirthday fails, because ringo isn’t in our BirthdayBook. This sets R8 to not_known and BD8 remains unbound.