

# Transformation Rules for Z

Mark Utting<sup>1</sup> and Petra Malik<sup>2</sup> and Ian Toyn

<sup>1</sup> Department of Computer Science  
The University of Waikato  
Hamilton, New Zealand,  
Email: marku@cs.waikato.ac.nz

<sup>2</sup> School of Mathematics, Statistics and Computer Science  
Victoria University  
Wellington, New Zealand,  
Email: petra@mcs.vuw.ac.nz

## Abstract

Z is a formal specification language combining typed set theory, predicate calculus, and a schema calculus. This paper describes an extension of Z that allows transformation and reasoning rules to be written in a Z-like notation. This gives a high-level, declarative, way of specifying transformations of Z terms, which makes it easier to build new Z manipulation tools. We describe the syntax and semantics of these rules, plus some example reasoning engines that use sets of rules to manipulate Z terms. The utility of these rules is demonstrated by discussing two sets of rules. One set defines expansion of Z schema expressions. The other set is used by the ZLive animator to preprocess Z expressions into a form more suitable for animation.

## 1 Introduction

The Z notation (ISO/IEC 13568 2002) is a formal specification language that combines typed set theory, predicate calculus, and a schema calculus. It has been widely used for the design and specification of computing systems (Hayes 1987, Woodcock & Davies 1996). For illustration purposes, consider the Z specification of a birthday book (Spivey 1992). We start by declaring a new Z section called *birthdaybook*:

**section** *birthdaybook* **parents** *standard\_toolkit*

By declaring *standard\_toolkit* as a parent, definitions for symbols like  $\leftrightarrow$  (partial function) are included for later use. The toolkit definitions have been standardised in (ISO/IEC 13568 2002), henceforth referred to as Standard Z.

Z specifications consist of a mixture of narrative text (like this paragraph) and formal Z paragraphs. The following Z paragraph introduces a set of names *NAME* and a set of dates *DATE* as new given types:

[*NAME*, *DATE*]

The *BirthdayBook* schema defines the state space of the birthday book system; the set of known names and a mapping from the set of names to the set of dates:

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the Fifteenth Computing: The Australasian Theory Symposium (CATS2009), Wellington, New Zealand. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 94, Rod Downey and Prabhu Manyem, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

$\begin{array}{l} \textit{BirthdayBook} \\ \textit{known} : \mathbb{P} \textit{NAME} \\ \textit{birthday} : \textit{NAME} \leftrightarrow \textit{DATE} \\ \textit{known} = \text{dom } \textit{birthday} \end{array}$
--

The initial state of the system is specified by the following schema:

$\begin{array}{l} \textit{Init} \\ \textit{BirthdayBook} \\ \textit{known} = \emptyset \end{array}$
---

It requires the set of known names to be empty. Including the schema *BirthdayBook* makes sure that the initial state is a valid state of the birthday book system. Using a schema as a declaration is short for declaring and constraining all the variables of the schema as given by the schema. That is, schema *Init* could have been written as:

$\begin{array}{l} \textit{InitExpanded} \\ \textit{known} : \mathbb{P} \textit{NAME} \\ \textit{birthday} : \textit{NAME} \leftrightarrow \textit{DATE} \\ \textit{known} = \text{dom } \textit{birthday} \\ \textit{known} = \emptyset \end{array}$
--

Schema *InitExpanded* is equivalent to the previously defined schema *Init*. This fact can formally be expressed using the Standard Z conjecture notation

$\vdash? \textit{Init} = \textit{InitExpanded}$

and a Z theorem prover could be used to prove it.

Next, we consider operations on the birthday book. The *AddBirthday* schema defines the non-error case of an operation for adding a new entry to the birthday book. It relates inputs (*name?* and *date?*), the pre-state (*known* and *birthday*), and the post-state (*known'* and *birthday'*). If the given name is not yet known, a mapping from the given name to the given date is added to the existing birthday book entries:

$\begin{array}{l} \textit{AddBirthday} \\ \Delta \textit{BirthdayBook} \\ \textit{name?} : \textit{NAME} \\ \textit{date?} : \textit{DATE} \\ \textit{name?} \notin \textit{known} \\ \textit{birthday}' = \textit{birthday} \cup \{\textit{name?} \mapsto \textit{date?}\} \end{array}$
--

The declaration  $\Delta \textit{BirthdayBook}$  is equivalent to a schema that includes both *BirthdayBook* and its decorated version *BirthdayBook'*:

$$\vdash? \Delta \text{BirthdayBook} = \\ [\text{BirthdayBook}; \text{BirthdayBook}' ]$$

$\text{BirthdayBook}'$  is schema  $\text{BirthdayBook}$  with its components primed:

$$\vdash? \text{BirthdayBook}' = \\ [ \text{known}' : \mathbb{P} \text{NAME}; \\ \text{birthday}' : \text{NAME} \leftrightarrow \text{DATE} \mid \\ \text{known}' = \text{dom } \text{birthday}' ]$$

Using these two equivalences, we can give an expanded equivalent definition of  $\text{AddBirthday}$ :

$\text{AddBirthdayExpanded}$
$\begin{aligned} &\text{known} : \mathbb{P} \text{NAME} \\ &\text{birthday} : \text{NAME} \leftrightarrow \text{DATE} \\ &\text{known}' : \mathbb{P} \text{NAME} \\ &\text{birthday}' : \text{NAME} \leftrightarrow \text{DATE} \\ &\text{name}? : \text{NAME} \\ &\text{date}? : \text{DATE} \end{aligned}$
$\begin{aligned} &\text{known} = \text{dom } \text{birthday} \\ &\text{known}' = \text{dom } \text{birthday}' \\ &\text{name}? \notin \text{known} \\ &\text{birthday}' = \text{birthday} \cup \{ \text{name}? \mapsto \text{date}? \} \end{aligned}$

$$\vdash? \text{AddBirthday} = \text{AddBirthdayExpanded}$$

Expansion of schemas as demonstrated above is an operation that is frequently performed by Z theorem provers, animators, translators, and other Z tools that transform Z specifications. However, in most Z tools, these and other Z transformations are performed by large amounts of low-level code (for example, in C or Java) that manipulate the syntax of Z schema expressions and return equivalent expanded terms. This low-level approach is error-prone, and the code that performs the transformations is time-consuming to write and difficult to read. The problem is that the core ideas of the abstract transformations are hidden in the masses of low-level code needed to manipulate syntax trees.

This means that building new Z transformation tools requires programming and debugging skills as well as a detailed knowledge of the API for manipulating Z syntax trees. In contrast, in an ideal world, it should be quick and easy to define new Z transformations by writing them in a high-level, declarative style that is easily understood and can be proven correct. Such a high-level notation gives better support for the capture and reuse of the knowledge implicit in the transformations (Armour 2000).

A natural approach would be to express transformations using theorems written in the Z language itself. The conjectures for the birthday book given above could be used in this way. However, many transformations of interest cannot be expressed using this approach. For example, what it means for a schema to be used as a declaration can be formalised for special instances like the  $\text{BirthdayBook}$  and  $\text{Init}$  schemas given above but cannot be written within the Z notation as a general rule that is applicable to arbitrary schemas.

In logic, rules are usually formulated as rule schemata that use special variables that can stand for arbitrary expressions, predicates, declarations etc. This paper proposes an extension of Standard Z called ZedRules that allows rule schemata using meta-level constraints to be written in a Z-like, declarative style. The ZedRules notation has been implemented and used in the CZT system (*Community Z Tools* 2008, Malik & Utting 2005).

Section 2 discusses approaches to schema expansion and user-defined transformations in existing Z tools. Section 3 describes the syntax and semantics of ZedRules. Section 4 describes three example reasoning engines provided by CZT, which apply the rules. Section 5 defines meta-level operations that we have found to be needed in ZedRules for reasoning about Z terms. Section 6 defines rewrite rules for expansion and normalisation of schema expressions, and Section 7 shows some rules that are used by a Z animator. Section 8 presents our conclusions.

## 2 Related Work

We briefly consider how other Z tools provide support for expanding schema expressions and for allowing users to define domain-specific transformations of Z expressions.

CADiZ (Toyn 2007) provides direct support for reasoning in Z (not indirectly via a theorem prover for another notation). It has a large fixed set of several hundred low-level inference rules, including rules for expanding individual schema operations and normalising individual declarations. A tactic language allows users to program compositions of inference rules and tactics to produce higher-level inference steps. The tactic language is based on ANGEL (Martin et al. 1996), to which has been added pattern matching using a Z-like syntax extended with jokers. Due to the inefficiency of the tactic interpreter, CADiZ provides little actual support for reasoning at a high level. One exception is a *schema expansion* tactic, which has been hand-coded in C for efficiency. This schema expansion tactic fully expands all schema operations in the Z term to which it is applied, normalising declarations as necessary.

Z/EVES (ORA Canada 1999) provides indirect support for reasoning in Z via the EVES prover, which is programmed in Common Lisp. It offers a small fixed set of high-level inference steps, including an unfold command for normalising schema expressions and a rewrite command that reasons according to rewrite rules. The effects of the rewrite command can be extended by specifying that some of the user's Z conjectures be used as rewrite rules, thus extending the rewrite command with domain-specific abilities. However, the genericity of these added rewrite rules is limited to the genericity of Z's generic theorems, which allows parametrization only over expressions, not other terms. Note that Z/EVES does not provide any low-level inference steps.

ProofPower (Arthan 2007) provides indirect support for reasoning in Z via an implementation of HOL. A wide range of high and low-level inference steps are provided, written in HOL's metalanguage SML. New inference steps can be defined by writing more SML. These inference steps can have premisses, jokers, etc. However, the SML notation for rules is not based on Z syntax.

Our ZedRules notation differs from the above approaches in aiming to express rules using Z syntax. In contrast, the other Z tools use a mixture of built-in commands, rewrite rules and specific tactics to implement schema expansion. ZedRules supports general inference rules with provisos, not just rewrite rules. They are written in a declarative style in a superset of Z. We shall see that this, combined with engines for applying rules in particular ways, is sufficiently expressive that we can write a set of rules for expanding schema expressions, which is one of the more complex aspects of Z. Our hope is that the expressivity of the ZedRules notation will make it easier to specify many kinds of Z transformations besides schema expansion.

Another difference between ZedRules and the approaches discussed above is the issue of soundness. The Z provers described above ensure that users do not introduce unsound inference steps (assuming that the prover itself is sound and bug-free). However, ensuring soundness of the rules written in the ZedRules notation is left to the rule writer. In Section 7 we will see that even unsound rules can be useful. Furthermore, having rules written in a Z-like notation rather than in low-level code makes them easier to verify. We consider below how to determine whether a particular rule is valid.

### 3 Rule Syntax and Semantics

This section describes the syntax and semantics of ZedRules. We add them as an extension to the Standard Z notation. The ideas in this section are not specific to CZT and could be adopted by other tools.

The following simple example rule, taken from the  $\mathcal{V}$  logic (Brien & Martin 2000), provides a first illustration of the ZedRule notion:

**section** *example*

★*Expr*  $E, F, Y$

**rule** *Funct*

$$\frac{\exists_1 x : F \bullet x.1 = E \quad x \setminus E}{Y = F(E) \Leftrightarrow (E, Y) \in F}$$

The rule says that if a relation  $F$  is functional at  $E$ , expressed in a way that is valid provided variable name  $x$  is not free in  $E$ , then an equality with an application of  $F$  is equivalent to a corresponding membership test. In this rule,  $x$  is an object-level name while  $E, F$ , and  $Y$  are meta-variables, called jokers in ZedRules. These jokers are declared in the ★*Expr* line above the rule, which says that these particular jokers can be used in place of expressions; we call them “expression jokers”.

In the  $\mathcal{V}$  logic, the not-free-in operator  $(- \setminus -)$  is an object-level relation, but in many other logics, not-free-in is a meta-level operator. Most logics contain rules with meta-level side conditions, also called provisos, which ensure various syntactic properties of the terms in the rule. ZedRules allows provisos to be written within a rule using object-level notation. However, the semantics of these provisos cannot be expressed within the Z notation. Oracle paragraphs are used to declare proviso notations as having meta-level interpretations. In the implementation, oracles are associated to code that checks these meta-level side-conditions.

#### 3.1 Syntax

Syntactically, a rule is a new kind of Z paragraph that starts with the keyword **rule** and is thus distinguishable from other Z paragraphs. This keyword is followed by a unique name for the rule. A rule has a single conclusion, written below a line, and zero or more premisses, written above the line. The syntax of consequents and premisses is that of Standard Z predicates extended with jokers.

The jokers must have their syntactic roles declared before they can be used. These syntactic roles are distinguished by keywords, including *Expr*, *ExprList*, *Pred*, *Name*, *NameList*, *DeclList*, *RenameList* and *Stroke*. The jokers used in the rest of this paper are declared as follows.

**section** *jokers*

★*DeclList*  $D, D1, D2, D3, D4, D5, D6, D7, D8, D9, D10$   
 ★*Pred*  $P, P1, P2, P3, P4, P5, P6, P7, P8, P9, P10$   
 ★*Expr*  $T, E, E1, E2, E3, E4, E5, E6, E7, E8, E9, E10$   
 ★*Name*  $N, N1, N2, N3, N4, N5, N6, N7, N8, N9, N10$   
 ★*NameList*  $NL$   
 ★*ExprList*  $EL$   
 ★*RenameList*  $RL$   
 ★*Stroke*  $S$

To distinguish premisses that are provisos from other premisses, we need to declare the proviso notations and associate them with corresponding meta-level interpretations. This is done using oracle paragraphs, which are so-called because we say that a proviso is interpreted by reference to an oracle. In an implementation, an oracle paragraph is associated with code that checks the corresponding property. An oracle paragraph should be accompanied by informal text that clarifies this meta-level interpretation.

An oracle paragraph starts with the keyword **oracle** followed by a unique name for the oracle followed by a Standard Z predicate extended with jokers. A premiss in a rule is a proviso if it matches the predicate of an oracle. For example, the not-free-in proviso is introduced by the following operator template and oracle paragraphs.

**relation**  $(- \setminus -)$

**oracle** *NotFreeIn*

$N \setminus E$

This oracle paragraph declares predicates of the form  $N \setminus E$ , where  $N$  and  $E$  are expression jokers, to be provisos. The second premiss of the example rule *Funct* above can thus be seen to be a proviso, distinguished by its use of the  $(- \setminus -)$  operator.

We use L<sup>A</sup>T<sub>E</sub>X mark-ups of rule, oracle and joker paragraphs that are distinct from those of Standard Z paragraphs, and hence the new paragraphs are treated as narrative text by other Z tools. This allows us to embed rules within Z specifications, but to retain backwards compatibility with Z tools that do not use the ZedRules notation.

#### 3.2 Type System

Typechecking rules and oracles involves typechecking their constituent predicates. This needs to differ from the standard typechecking of predicates only in the case of names. Names can be partitioned according to what they refer to: references to joker declarations; references to declarations quantified within predicates; and references to global declarations of the specification, which are said to be free in the rule.

The scope of a joker currently encompasses all rule and oracle paragraphs of the section in which it appears and of which it is an ancestor. In the future, we plan to declare jokers locally to a rule or oracle paragraph instead. Global jokers are very concise (need to be declared just once) but are less flexible because once a joker is declared, its name becomes unusable as an ordinary Z name throughout its global scope. Worse, if a joker is renamed (for whatever reason), then it can be difficult to be sure that all references to the joker have been renamed.

Within a rule or oracle paragraph, a name refers to a joker declaration if there is a joker with that name, and otherwise to a quantified declaration if there is one with that name, otherwise the name is free.

The scope rule for free names should be consistent with the desired effects of applying rules. A rule might be applied to a term within a conjecture paragraph, or within a goal arising from a conjecture, in the course of some reasoning such as a proof. Or a rule

might be applied as an equivalence transformation to a term in some other paragraph of the specification. Consider applying the example rule above to rewrite the equality in the following conjecture.

**section** *scope\_example* **parents** *standard\_toolkit*

$\vdash? \exists n : \mathbb{N} \bullet n = succ\ 2$

Instantiating the rule's  $Y$  with  $n$ ,  $F$  with  $succ$  and  $E$  with  $2$ , the desired effect is to rewrite the equality to  $(2, n) \in succ$ . So the instantiations of the jokers may need to refer to declarations whose scope is locally quantified within the goal ( $n$ ), but we see no need for free names in the rule ever to refer to such local declarations. This conveniently allows the typechecking of free names to be based solely on the environment where the rule is written.

If a rule is written in a different  $Z$  section from a paragraph to which the rule is applied, then the environments of top-level declarations for these two paragraphs could provide different declarations of the rule's free names. (In the case of a goal in a proof, the environment is assumed to be the same as that of the conjecture being proved.) We should ensure that the declaration referred to by a free name in a rule is in scope for the paragraph to which the rule is applied. This is the case if the rule is in the same  $Z$  section as, or is in a  $Z$  section that is a parent or older ancestor of, the paragraph of the term to which the rule is applied. This cannot be checked until a rule application is attempted, but it requires only one check per rule application rather than one check per free name.

Typechecking a rule can result in constraints being inferred between the types of distinct jokers. In the case of the example rule above, type constraints can be inferred between  $E$ ,  $F$  and  $Y$ : there exist types  $T1, T2$  such that  $E : T1$ ,  $Y : T2$  and  $F : \mathbb{P}(T1 \times T2)$ . When a rule is applied, particular instantiations are determined (in this example,  $T1$  and  $T2$  are both instantiated to the numeric base type,  $\mathbb{A}$ ) and these type constraints should then be checked. Equivalently, one can typecheck the whole of the instantiated rule, though this is less efficient.

### 3.3 Provisos

Provisos express useful meta-level properties of the terms in a rule, such as properties of signatures, and distinguishing different decorations on names. The evaluation of a proviso takes into account any known instantiations of jokers. If no jokers remain, the proviso acts as a true/false check. Otherwise, an attempt is made to calculate deterministically particular instantiations of the remaining jokers to make the proviso be true.

$Z$  does not provide any object level syntax for types and signatures, which is unfortunate for provisos that express properties of types and signatures. This is made to work by requiring that types be written using expression notation. For example, we have a typecheck oracle defined as follows.

**section** *oracle\_example*  
**parents** *standard\_toolkit, jokers*

**relation** ( $_ :: _$ )

$\frac{[E1, E2]}{_ :: _ : E1 \leftrightarrow E2}$

**oracle** TypecheckOracle  
 $E :: T$

The meta-level interpretation of the notation  $E :: T$  is true provided  $T$  is the carrier set of  $E$ . Note that  $T$  is an expression joker, yet stands for a type. We often write  $\mathbb{P}[D \mid true]$  for the type of a schema in which the signature is written as the declarations  $D$ .

In the ZedRules implementation, each oracle is associated to code that checks the corresponding meta-level property. For the *TypecheckOracle* given above, for example, there is code that takes two expressions  $E$  and  $T$  as arguments and returns true if  $T$  is the carrier set of  $E$ . If  $E$  and  $T$  are fully instantiated (do not contain any jokers), this is decidable and indeed easily checked. If  $E$  does not contain uninstantiated jokers but  $T$  does, the carrier set for  $E$  can be computed and matched against  $T$  to determine instantiations for those jokers.

This evaluation of a proviso must be monotonic with respect to the instantiation of jokers. For example, given the proviso  $E :: T$ , it is not acceptable for the proviso to evaluate to *true* when  $E$  is a joker, but evaluate to *false* when  $E$  is instantiated. In fact, this property implies that when a proviso  $E :: T$  contains a joker in  $E$ , either the proviso must evaluate to *unknown* or *false*, or the proviso must add constraints to the joker to ensure that it can never be instantiated with an expression whose carrier set is not  $T$ . The CZT implementation of ZedRules takes the former approach. The latter approach is more powerful, but also more complex to implement.

### 3.4 Rule Semantics

We describe the semantics of ZedRules independently of any operational description of how sets of rules can be used. This allows rules to be used by many different reasoning and rewrite engines. Each engine may place different restrictions on the rules that it allows, may apply rules using a variety of operational semantics (apply once, apply exhaustively, apply to all sub-terms bottom-up or top-down etc.), and may use a different implementation technology (for example, an interpreter that applies the rules or a compiler that transforms a set of rules into code).

A well-typed rule is a template in which the jokers can be instantiated to particular well-typed terms to give instances of the rule. The semantics of a well-typed rule concerns whether all instances of the rule are valid. A rule is valid if it could not be used to derive a false conclusion from true premisses when all of its provisos are true. In other words, for all well-typed instantiations, provided all the provisos are true, the instantiated other premisses should imply the instantiated consequent. Using meta-notation based on that of the ISO  $Z$  standard, a rule with instantiated non-proviso premisses  $p_1, \dots, p_n$  and instantiated consequent  $p$  is valid if

$$\llbracket p_1 \wedge \dots \wedge p_n \Rightarrow p \rrbracket^P \supseteq \llbracket z \rrbracket^z i$$

where  $z$  is the whole  $Z$  specification, and  $i$  is the name of the section where the rule is defined. This says that the set of models in which the conjunction of the premisses implies the consequent contains all the models of the section in which the rule is defined.

The meta-level interpretation of provisos has the effect of excluding (or filtering out) instantiations for which the rule might not be valid. Any object-level constraints on proviso notations have no effect on rule semantics.

The instances of a rule that are used in practice are determined by where the rule is applied. The rule applications must be in contexts where the rule's free names refer to the same definitions as in the context where the rule is defined. This can be checked by ensuring that the section where the rule is defined is

the same as, or is an ancestor of, the section where the rule is applied, thanks to the scope rules of Standard Z.

The validity of a rule depends on its premisses and consequent being in the relationship specified above. We expect the person who writes the rule to ensure that this is the case. Our tools check the applicability of a rule. Given valid rules and valid applications of rules, compositions of rule applications are valid, and hence valid derived rules can be deduced involving any remaining premisses.

## 4 Tools for ZedRules

To use and test ZedRules, several tools have been implemented in the CZT system: a rewrite engine, an automated depth-first prover, and an interactive prover. Note that none of these is intended to be a general Z theorem prover, for the following reasons.

Firstly, there are already many other good theorem provers for Z, so developing another would simply fragment the existing small user-base further.

Secondly, for practical theorem proving, it is necessary to develop a large set of rules (several for each Z construct), plus many derived rules and tactics that automate the common styles of reasoning. Experience shows that this requires several person years of theory development.

Finally, the current CZT implementation is fast enough for simple unfolding and expansion tasks (a few hundred rules per second), but would need to be significantly faster if we were to use it for heavy-duty reasoning with lots of rules.

### 4.1 A Rewrite Engine

CZT provides a rewrite engine that can rewrite a given Z term using a given set of rewrite rules. A rewrite rule is a rule whose conclusion is of the form  $t1 \ r \ t2$ , where the left side  $t1$  must match the term to be rewritten, the right side  $t2$  represents the result of the transformation, and  $r$  is some equivalence relation. Currently supported are the rewriting of expressions (using  $e1 = e2$  rules) and predicates (using  $p1 \Leftrightarrow p2$  rules).

Terms can be rewritten just once or combined into sequences of rewrites. For the rewrite rules presented in this paper, a leftmost innermost traversal function is used (van den Brand et al. 2003). Sub-terms are rewritten in a left-to-right, bottom-up fashion until no more rewrite rules can be applied. The rule writer needs to ensure termination of the rules.

Note that rewrite rules may have premisses, so applying such rewrite rules requires proving any premiss predicates and provisos. This task is done using the automated prover described next.

### 4.2 An Automated Depth-First Prover

The automated prover uses a depth-first search to find proofs automatically. Given a predicate, it tries to apply the rules in the order they appear in the Z specification. If the conclusion of a rule can be matched with the predicate, the resulting joker instantiations are recorded and a sequence of new sub-goals is created from the premisses of the rule (taking the joker instantiations into account) and attempted to be proven next (in the order provided by the rule). If proving the new subgoals fails, the other rules are tried instead. To discharge a proviso, code for checking the predicate is executed. If this succeeds, the predicate is considered to be proven.

Note that this prover uses a shallow backtracking algorithm and therefore might fail to find a proof

even if one exists. If a proof for a subgoal is found, the prover sticks to it and no attempt is made to find an alternative proof. The joker instantiations associated with this particular proof might prevent another subgoal from being proven and thus might prevent the prover from finding a proof. Even though this approach is less general than deep backtracking, this has not been a problem for the rules used so far. Furthermore, if a proof is found using shallow backtracking, it is usually found more quickly than it would be using deep backtracking.

### 4.3 An Interactive Prover

The interactive prover allows a human user to apply rules and oracles individually, and hence be in full control of their application. Although this is an inefficient and tedious way of finding proofs, it has been found to be a useful tool in the development of rule sets, such as the rule sets that are used by the tools discussed in the preceding sections.

The interactive prover displays a proof in tree form, as illustrated in Figure 1. The top line is the root of the tree, and shows the conjecture being proved: that *BirthdayBook* could be expanded to the given schema. Indentation shows that the next two lines are sub-goals; they arose from choosing a rule to be applied to the goal at the root. Both of these sub-goals are instantiated provisos of the applied rule. The menu lists the names of commands that could be chosen to be applied to the selected sub-goal. Choosing to apply an oracle generates a further node in the tree, as the chosen oracle might have to wait for evaluation of another oracle before having sufficient information to make its decision. The other sub-goal has already been proved by choosing an oracle and then proving it using that oracle. The icons indicate whether a node has been proved (‘✓’) or not (‘?’). After a node has been proved, the name of the rule or oracle that was used to prove it is displayed as a tooltip whenever the cursor is positioned over the node. Sub-trees can be folded away to save screen space.

More generally, the pop-up menu offers commands for the application of any applicable rule or oracle, and undoing rule and proviso applications, as appropriate to the selected node. The “Auto prove” menu entry calls the automated prover described in the previous section. If a proof is found, the resulting proof tree is displayed as a subtree of the current node, and the user can browse and manipulate it. The “Why not” submenu lists all the rules that cannot be applied. By selecting one of those, the user is presented with detailed debugging information about why the matching of the selected goal’s predicate with the conclusion of a particular rule failed.

To provide the submenus “Apply” and “Why not”, the prover goes through all the known rules and tries to apply them to the current predicate. If one of these applications succeeds, the corresponding rule name is added into the “Apply” submenu; if it fails, the name is added into the “Why not” submenu. While this approach might be too inefficient and slow for large rule sets, it has been very helpful for our purposes.

## 5 Oracles

The oracles that we have found useful so far are defined in *oracle\_toolkit*.

```
section oracle_toolkit
parents standard_toolkit, jokers
```

This toolkit introduces the notation of provisos using Z operator templates. No Z constraints are given on

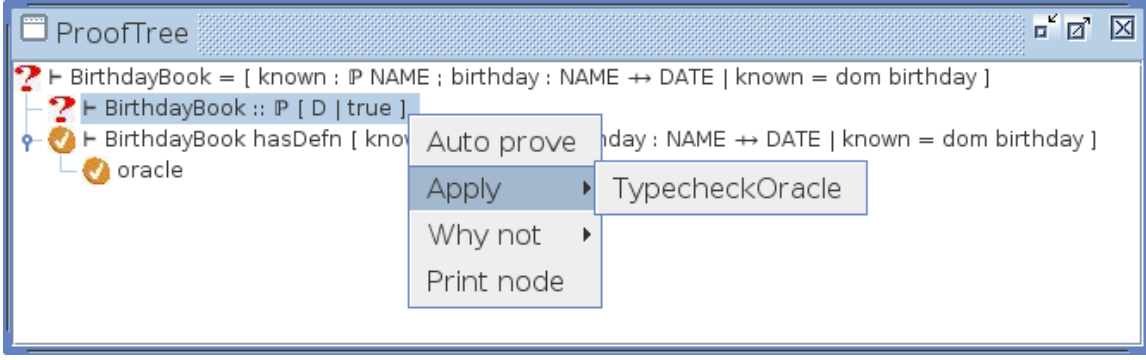


Figure 1: Screenshot of Interactive Prover

those operators; instead, oracle paragraphs are given to associate the operators with meta-level interpretations. This allows a premiss that matches the form of a proviso to be interpreted specially, while other premisses are merely matched.

**relation** (  $- :: -$  )

$$\frac{}{\frac{}{\frac{}{- :: - : E1 \leftrightarrow E2}}}}$$

**oracle** TypecheckOracle  
 $E :: T$

The meta-level interpretation of the notation  $E :: T$  is true provided  $T$  is the carrier set of  $E$ . This works for generic as well as non-generic types: if the type of  $E$  depends on a generic parameter type, then expression  $T$  contains a reference to the corresponding generic parameter.

**relation** (  $- \text{ hasDefn } -$  )

$$\frac{}{\frac{}{\frac{}{- \text{ hasDefn } - : \mathbb{P}(E \times E)}}}}$$

**oracle** LookupOracle  
 $E1 \text{ hasDefn } E2$

$E1 \text{ hasDefn } E2$  is true provided  $E1$  is either a reference expression or a generic instantiation expression instantiated with names, and the referenced definition exists in the environment and has the form  $E1 : E2$ . It provides a way of looking up definitions so that references can be expanded, and can also be used to check that a name is defined in a particular way. No way is yet provided to look-up a definition's associated constraints, nor have we provided a way to look-up a definition of the form  $E1 == E2$ .

**relation** (  $- \text{ is } -$  )

$$\frac{}{\frac{}{\frac{}{- \text{ is } - : \mathbb{P}(E1 \times E2)}}}}$$

**oracle** DecorOracle  
 $E^S \text{ is } E2$

$E^S \text{ is } E2$  is true provided  $E$  and  $E2$  are schemas and decorating  $E$  with stroke  $S$  results in  $E2$ . Given  $S$  and either  $E$  or  $E2$ , the other can be calculated.

**oracle** ThetaOracle  
 $\theta E \text{ is } E2$

$\theta E \text{ is } E2$  is true provided that the binding constructed by  $\theta E$  is the same as the binding extension  $E2$ .  $E2$  can be calculated deterministically from  $E$ , but not  $E$  from  $E2$ .

**oracle** DecorThetaOracle  
 $\theta E^S \text{ is } E2$

$\theta E^S \text{ is } E2$  is true provided that the binding constructed by  $\theta E^S$  is the same as the binding extension  $E2$ . Given  $S$ ,  $E2$  can be calculated deterministically from  $E$ , but not  $E$  from  $E2$ .

**oracle** SchemaMinusOracle  
 $[D1 \mid \text{true}] \setminus [D2 \mid \text{true}] \text{ is } [D3 \mid \text{true}]$

$[D1 \mid \text{true}] \setminus [D2 \mid \text{true}] \text{ is } [D3 \mid \text{true}]$  is true provided that subtracting the signature  $D2$  from the signature  $D1$  results in the signature  $D3$ .

**oracle** UnprefixOracle  
 $N1 \text{ unprefix } N2 \text{ is } N3$

$N1 \text{ unprefix } N2 \text{ is } N3$  is true provided that name  $N2$  has the characters of name  $N1$  as a prefix, which when removed leaves the name  $N3$ .

**oracle** SplitNamesOracle  
 $(\text{split}[D \mid \text{true}]) \text{ is } ([D1 \mid \text{true}]? \wedge [D2 \mid \text{true}] \wedge [D3 \mid \text{true}]' \wedge [D4 \mid \text{true}]!)$

$(\text{split}[D \mid \text{true}]) \text{ is } ([D1 \mid \text{true}]? \wedge [D2 \mid \text{true}] \wedge [D3 \mid \text{true}]' \wedge [D4 \mid \text{true}]!)$  is true provided that partitioning the declarations of signature  $D$  according to their rightmost stroke results in the same declarations as decorating the declarations of signatures  $D1$ ,  $D2$ ,  $D3$  and  $D4$  with appropriate strokes. Note that  $D2$  concerns only those declarations with no strokes. For example,  $(\text{split}[x, x', y?, y!, z : T]) \text{ is } ([y : T]? \wedge [x : T; z : T] \wedge [x : T]' \wedge [y : T]!)$ .

**oracle** HideOracle  
 $[D1 \mid \text{true}] \setminus (NL) \text{ is } \exists [D2 \mid \text{true}] \bullet [D1 \mid \text{true}]$

$[D1 \mid \text{true}] \setminus (NL) \text{ is } \exists [D2 \mid \text{true}] \bullet [D1 \mid \text{true}]$  is true provided that the signature  $D2$  declares the same names as the list of names  $NL$  with the same types as they are declared in  $D1$ .

**oracle** RenameOracle  
 $E[RL] \text{ is } E2$

$E[RL] \text{ is } E2$  is true provided that the renaming of schema  $E$  is the same as schema  $E2$ .

**oracle** XiOracle

$$(\theta[D \mid \text{true}] = \theta[D \mid \text{true}]') \Leftrightarrow P$$

$(\theta[D \mid \text{true}] = \theta[D \mid \text{true}]') \Leftrightarrow P$  is true provided that the truth of the predicate from a  $\Xi$  schema is equivalent to the conjunction of equalities  $P$ .  $P$  can be calculated deterministically from  $D$ , but not  $D$  from  $P$ .

## 6 Example Rules: Schema Expansion and Normalisation

This section contains rewrite rules for expanding and normalising declarations and schema expressions. For example, given a schema expression such as  $E \wedge [x : \mathbb{N} \mid p \ x]$  where  $E$  is the name of a schema, these rules define an expansion and normalisation process that will transform this schema expression into an equivalent expression of the form  $[D \mid P]$ , where  $D$  is a list of variable declarations whose types are all carrier sets and  $P$  is a predicate that will contain constraints like  $x \in \mathbb{N}$  and  $p \ x$ . There is one rule for each schema operator, with most requiring that the operands be already expanded, hence the rewrite engine usually expands schema expressions in order from the innermost to the outermost.

### 6.1 Normalisation of Declaration Lists

**section** *normalisation\_rules*  
**parents** *oracle\_toolkit*

This section is concerned with making explicit the constraints that may be implicit in the declarations of a schema construction expression. A relation *pred* is introduced to indicate where this is wanted, then rules are given to rewrite applications of *pred*.

The rules recurse through a declaration list from left to right, with the base case of an empty declaration list being handled by the *PredEmptyDecl* rule. We assume that multiple declarations such as  $x, y, z : T$  are expanded out into separate declarations before rules are applied, so the rules that follow cover all possible kinds of declarations.

The *PredVarDecl1* rule is a special case of *PredVarDecl2*. It applies when the expression  $E$  is already a carrier set. Since *PredVarDecl1* comes before *PredVarDecl2* in this section, the rewrite engine (see Section 4.1) gives it higher priority, and this avoids introducing redundant tautologies (such as  $E \in \mathbb{A}$ , which is guaranteed to be true by the type system) into the predicate. This is an example of how we can influence the behaviour of the rewrite engine by placing more specific rules before more general ones. Of course, in the interactive prover, the user could choose to apply either rule when  $E$  is a carrier set.

**relation** (*pred* \_)

**rule** *PredVarDecl1*

$$\frac{E :: \mathbb{P} \ E}{\text{pred}[N : E; D \mid \text{true}] \Leftrightarrow \text{pred}[D \mid \text{true}]}$$

**rule** *PredVarDecl2*

$$\text{pred}[N : E; D \mid \text{true}] \Leftrightarrow N \in E \wedge \text{pred}[D \mid \text{true}]$$

**rule** *PredConstDecl*

$$\text{pred}[N == E; D \mid \text{true}] \Leftrightarrow N = E \wedge \text{pred}[D \mid \text{true}]$$

**rule** *PredIncludeDecl*

$$\text{pred}[E; D \mid \text{true}] \Leftrightarrow E \wedge \text{pred}[D \mid \text{true}]$$

**rule** *PredEmptyDecl*

$$\text{pred}[\mid \text{true}] \Leftrightarrow \text{true}$$

### 6.2 Expansion of Schema Expressions

**section** *expansion\_rules*  
**parents** *normalisation\_rules*

This section defines the expansion of schema expressions.

The Theta rule rewrites a binding construction expression to a binding extension expression. The type proviso is used to calculate the signature, from which the theta proviso generates the appropriate binding extension expression.

**rule** Theta

$$\frac{E :: \mathbb{P}[D \mid \text{true}] \quad \theta[D \mid \text{true}] \text{ is } E2}{\theta E = E2}$$

For example, if  $E$  is  $\theta \text{ BirthdayBook}$ , then  $D$  is calculated to be

$$\text{known} : \mathbb{P} \ \text{NAME}; \ \text{birthday} : \mathbb{P}(\text{NAME} \times \text{DATE})$$

and then  $E2$  is calculated to be

$$\langle \! \langle \text{known} == \text{known}, \text{birthday} == \text{birthday} \! \rangle \! \rangle.$$

The next rule handles decorated  $\theta$  expressions. This rule handles only one stroke. In the future, we intend to add support for Stroke-List jokers, so that all decorations can be handled by this rule.

**rule** *ThetaDecor*

$$\frac{E :: \mathbb{P}[D \mid \text{true}] \quad \theta[D \mid \text{true}]^S \text{ is } E2}{\theta E^S = E2}$$

The schema decoration rule has a proviso to calculate the effect of the decoration.

**rule** *SchemaDecor*

$$\frac{[D1 \mid P1]^S \text{ is } [D2 \mid P2]}{[D1 \mid P1]^S = [D2 \mid P2]}$$

The schema renaming rule has a proviso to calculate the effect of the renaming.

**rule** *SchemaRenaming*

$$\frac{E[RL] \text{ is } E2}{E[RL] = E2}$$

Schema negation involves normalising the declarations and negating the resulting predicate.

**rule** *SchemaNegation*

$$\frac{[D \mid P] :: \mathbb{P}[D1 \mid \text{true}]}{(\neg [D \mid P]) = [D1 \mid \neg (\text{pred}[D \mid \text{true}] \wedge P)]}$$

Schema conjunction involves merging its operands, and hence may need them to be normalised; it's easiest to normalise them in all cases. Schema disjunction, implication and equivalence all require normalisation of their operands, and hence their rules are similar to the rule for schema conjunction.

$$\begin{array}{c}
\text{rule SchemaConjunction} \\
\frac{([D1 \mid true] \wedge [D2 \mid true]) :: \mathbb{P}[D3 \mid true]}{([D1 \mid P1] \wedge [D2 \mid P2]) = [D3 \mid \text{pred}[D1 \mid true] \wedge P1 \wedge \text{pred}[D2 \mid true] \wedge P2]}
\end{array}$$

$$\begin{array}{c}
\text{rule SchemaDisjunction} \\
\frac{([D1 \mid true] \vee [D2 \mid true]) :: \mathbb{P}[D3 \mid true]}{([D1 \mid P1] \vee [D2 \mid P2]) = [D3 \mid (\text{pred}[D1 \mid true] \wedge P1) \vee (\text{pred}[D2 \mid true] \wedge P2)]}
\end{array}$$

$$\begin{array}{c}
\text{rule SchemaImplication} \\
\frac{([D1 \mid true] \wedge [D2 \mid true]) :: \mathbb{P}[D3 \mid true]}{([D1 \mid P1] \Rightarrow [D2 \mid P2]) = [D3 \mid (\text{pred}[D1 \mid true] \wedge P1) \Rightarrow (\text{pred}[D2 \mid true] \wedge P2)]}
\end{array}$$

$$\begin{array}{c}
\text{rule SchemaEquivalence} \\
\frac{([D1 \mid true] \wedge [D2 \mid true]) :: \mathbb{P}[D3 \mid true]}{([D1 \mid P1] \Leftrightarrow [D2 \mid P2]) = [D3 \mid (\text{pred}[D1 \mid true] \wedge P1) \Leftrightarrow (\text{pred}[D2 \mid true] \wedge P2)]}
\end{array}$$

Schema projection is also similar to schema conjunction, but the resulting schema has only the names that are declared in the right operand. Any other names that are declared in the left operand are hidden by existential quantification.

$$\begin{array}{c}
\text{rule SchemaProjection} \\
\frac{([D1 \mid true] \wedge [D2 \mid true]) :: \mathbb{P}[D3 \mid true] \quad [D2 \mid true] :: \mathbb{P}[D4 \mid true] \quad ([D3 \mid true] \sphericalangle [D4 \mid true]) \text{ is } [D5 \mid true]}{[D1 \mid P1] \upharpoonright [D2 \mid P2] = [D2 \mid (\exists D5 \bullet \text{pred}[D1 \mid true] \wedge P1 \wedge P2)]}
\end{array}$$

Schema precondition involves hiding all output and next state components, such as  $x!$  and  $x'$ . The declarations whose names have no decorations or a final decoration of  $?$  are collected in  $D6$ .

$$\begin{array}{c}
\text{rule SchemaPrecondition} \\
\frac{[D \mid true] :: \mathbb{P}[D1 \mid true] \quad (\text{split}[D1 \mid true]) \text{ is } ([D2 \mid true]? \wedge [D3 \mid true] \wedge [D4 \mid true]' \wedge [D5 \mid true]!) \quad ([D2 \mid true]? \wedge [D3 \mid true]) :: \mathbb{P}[D6 \mid true]}{\text{pre}[D \mid P] = [D6 \mid (\exists ([D4 \mid true]' \wedge [D5 \mid true]!) \bullet \text{pred}[D \mid true] \wedge P)]}
\end{array}$$

Schema existential quantification involves calculating the declarations that are not hidden, and making explicit the constraints implicit in the original declarations.

$$\begin{array}{c}
\text{rule SchemaExists} \\
\frac{[D1 \mid true] :: \mathbb{P}[D3 \mid true] \quad [D2 \mid true] :: \mathbb{P}[D4 \mid true] \quad ([D4 \mid true] \sphericalangle [D3 \mid true]) \text{ is } [D5 \mid true]}{(\exists D1 \mid P1 \bullet [D2 \mid P2]) = [D5 \mid (\exists D1 \bullet P1 \wedge \text{pred}[D2 \mid true] \wedge P2)]}
\end{array}$$

The rule for schema unique existential quantification is analogous to that for schema existential quantification.

$$\begin{array}{c}
\text{rule SchemaExists1} \\
\frac{[D1 \mid true] :: \mathbb{P}[D3 \mid true] \quad [D2 \mid true] :: \mathbb{P}[D4 \mid true] \quad ([D4 \mid true] \sphericalangle [D3 \mid true]) \text{ is } [D5 \mid true]}{(\exists_1 D1 \mid P1 \bullet [D2 \mid P2]) = [D5 \mid (\exists_1 D1 \bullet P1 \wedge \text{pred}[D2 \mid true] \wedge P2)]}
\end{array}$$

The rule for schema universal quantification is also analogous to that for schema existential quantification.

$$\begin{array}{c}
\text{rule SchemaForall} \\
\frac{[D1 \mid true] :: \mathbb{P}[D3 \mid true] \quad [D2 \mid true] :: \mathbb{P}[D4 \mid true] \quad ([D4 \mid true] \sphericalangle [D3 \mid true]) \text{ is } [D5 \mid true]}{(\forall D1 \mid P1 \bullet [D2 \mid P2]) = [D5 \mid (\forall D1 \bullet P1 \wedge \text{pred}[D2 \mid true] \wedge P2)]}
\end{array}$$

Schema composition is one of the most complex schema operators. Nevertheless, this SchemaComposition rule captures its semantics concisely. First, the signatures of the two expressions  $E1$  and  $E2$  are computed, and then the names that appear primed in  $D1$  are collected in  $D5$ . The names that appear both primed in  $D1$  and unprimed in  $D2$  are computed by subtracting from  $D5$  those names that do not appear in  $D2$ . Note that we currently use the rarely used decoration  $\circ$ . In the future, we intend to consider using a special decoration that cannot be entered by users, rather like  $\bowtie$  (ISO/IEC 13568 2002).

$$\begin{array}{c}
\text{rule SchemaComposition} \\
\frac{E1 :: \mathbb{P}[D1 \mid true] \quad E2 :: \mathbb{P}[D2 \mid true] \quad (\text{split}[D1 \mid true]) \text{ is } ([D3 \mid true]? \wedge [D4 \mid true] \wedge [D5 \mid true]' \wedge [D6 \mid true]!) \quad ([D5 \mid true] \sphericalangle [D2 \mid true]) \text{ is } E3 \quad ([D5 \mid true] \sphericalangle E3) \text{ is } E4}{E1 \circ E2 = (\exists E4 \circ \bullet (\exists E4' \bullet [E1 \mid \theta E4' = \theta E4 \circ]) \wedge (\exists E4 \bullet [E2 \mid \theta E4 = \theta E4 \circ]))}
\end{array}$$

Schema piping is expanded in an analogous way to schema composition.

$$\begin{array}{c}
\text{rule SchemaPiping} \\
\frac{E1 :: \mathbb{P}[D1 \mid true] \quad E2 :: \mathbb{P}[D2 \mid true] \quad (\text{split}[D1 \mid true]) \text{ is } ([D3 \mid true]? \wedge [D4 \mid true] \wedge [D5 \mid true]' \wedge [D6 \mid true]!) \quad (\text{split}[D2 \mid true]) \text{ is } ([D7 \mid true]? \wedge [D8 \mid true] \wedge [D9 \mid true]' \wedge [D10 \mid true]!) \quad ([D6 \mid true] \sphericalangle [D7 \mid true]) \text{ is } E3 \quad ([D6 \mid true] \sphericalangle E3) \text{ is } E6}{E1 \gg E2 = (\exists E6 \circ \bullet (\exists E6! \bullet [E1 \mid \theta E6! = \theta E6 \circ]) \wedge (\exists E6? \bullet [E2 \mid \theta E6? = \theta E6 \circ]))}
\end{array}$$

Schema hiding involves existentially quantifying declarations whose names are those to be hidden and whose types are as in the original schema.

**rule** SchemaHiding

$$\frac{E :: \mathbb{P}[D1 \mid true] \quad [D1 \mid true] \setminus (NL) \text{ is } \exists[D2 \mid true] \bullet [D1 \mid true]}{E \setminus (NL) = \exists[D2 \mid true] \bullet E}$$

The GenInst rule is applicable to a generic instantiation provided the referenced definition is a schema. It uses the look-up proviso to determine the instantiated definition.

**rule** GenInst

$$\frac{N[EL] :: \mathbb{P}[D \mid true] \quad N[EL] \text{ hasDefn } E2}{N[EL] = E2}$$

The Ref rule is applicable to a reference expression provided the referenced definition is a schema. It uses the look-up proviso to determine the definition.

**rule** Ref

$$\frac{N :: \mathbb{P}[D \mid true] \quad N \text{ hasDefn } E2}{N = E2}$$

If the specification has an explicit definition of a  $\Delta$  schema, then the Ref rule expands its definition. But if the  $\Delta$  schema is implicitly defined, then the DeltaRef rule generates the implicit definition.

**rule** DeltaRef

$$\frac{\Delta \text{ unprefix } N \text{ is } N2}{N = [N2; N2']}$$

Similarly, the XiRef rule generates the implicit definition of a  $\Xi$  schema. It uses a proviso to simplify the constraint to a conjunction of equalities.

**rule** XiRef

$$\frac{\Xi \text{ unprefix } N \text{ is } N2 \quad [N2 \mid true] :: \mathbb{P}[D2 \mid true] \quad \theta[D2 \mid true] = \theta[D2 \mid true]' \Leftrightarrow P}{N = [N2; N2' \mid P]}$$

If a schema expression  $[D \mid P]$  is used as a predicate, it is equivalent to the predicate  $P$  conjoined with the constraints that are implicit in the declarations.

**rule** SchemaPred

$$[D \mid P] \Leftrightarrow (pred[D \mid true] \wedge P)$$

## 7 Example Rules: ZLive Preprocessor

In this section, we show just a few of the rules that the ZLive animator uses to expand and simplify Z expressions before starting animation. When an expression is given to ZLive to animate, the first phase of animation (after parsing and typechecking) is to use the CZT Rewrite Engine (see Section 4.1) to apply the following set of rules to the expression.

**section** ZLivePreprocess

**parents** *expansion\_rules*

Note that this *ZLivePreprocess* section has the *expansion\_rules* section (see Section 6) as a parent, so the rules from the *expansion\_rules* section are effectively the first rules in the *ZLivePreprocess* section. If we wanted to reuse multiple rule sets, we could simply list multiple parents, in the desired order.

The SchemaToSet rule rewrites schemas that are used as expressions to set comprehensions. Since

this rule appears *after* the schema rule from the *expansion\_rules* section, schema expressions will be expanded and normalised before this rule is applied.

**rule** SchemaToSet

$$[D \mid P] = \{D \mid P \bullet \theta[D \mid true]\}$$

The next two rules are adapted from the sets toolkit and the relations toolkit of Standard Z. They illustrate how rewrite rules can be used to unfold one operator and express it in terms of other simpler operators. ZLive needs no built-in knowledge of intersection or dom — these two rules are all that is needed. Around forty of the Standard Z toolkit operators are unfolded by rules in this way.

**rule** intersection

$$E1 \cap E2 = \{x : E1 \mid x \in E2\}$$

**rule** domain

$$\text{dom } E = \{x : E \bullet x.1\}$$

We finish this section with some rules that are useful despite being invalid. ZLive uses an internal data structure, RelSet, to represent function and relation spaces, such as  $\mathbb{N} \rightarrow \{0, 1\}$ . This data structure records the domain and range set, plus various flags according to the properties of the function space (total, onto, functional, injective). This gives a compact representation of the function space and makes it easy to check efficiently whether a given set of pairs is a member of the function space or not. The following two rules illustrate how the Z function space operators are rewritten into **let** terms, which ZLive recognises and converts into a RelSet object. These rules are not true equalities, but are used to rewrite a term like  $\mathbb{N} \rightarrow \{0, 1\}$  into the temporary **let** form, which is then translated into a RelSet object that is equivalent to  $\mathbb{N} \rightarrow \{0, 1\}$ .

**rule** fun

$$E1 \rightarrow E2 = (\text{let } isFun == 1; isTotal == 1 \bullet \mathbb{P}(E1 \times E2))$$

**rule** bij

$$E1 \rightsquigarrow E2 = (\text{let } isFun == 1; isTotal == 1; isOnto == 1; isInj == 1 \bullet \mathbb{P}(E1 \times E2))$$

For example, rule fun rewrites  $\mathbb{N} \rightarrow \{0, 1\}$  to

$$(\text{let } isFun == 1; isTotal == 1 \bullet \mathbb{P}(\mathbb{N} \times \{0, 1\}))$$

and this is translated into the RelSet object

$$\{f : \mathbb{P}(\mathbb{N} \times \{0, 1\}) \mid f \in \mathbb{N} \leftrightarrow \{0, 1\} \wedge \text{dom } f = \mathbb{N}\}$$

within ZLive, which is equivalent to the original input expression. Using rules to preprocess expressions and expand schemas has made it far faster to develop ZLive and makes it easier to experiment with alternative translations of an operator.

## 8 Conclusions

We have described the ZedRules notation and illustrated its utility for schema expansion and general rewriting. The main advantage of the notation is that it is a superset of Standard Z, which makes it much easier and clearer to develop Z tools that perform non-trivial transformations. We found a small

number of meta-level constraints that we could not express in Z notation and hence we defined oracles for those. These oracles are all concerned with the core Z language, for which formal semantics is given in (ISO/IEC 13568 2002).

In the future, we would like to be able to extend the ZedRules notation to support a variety of Z logics, such as  $Z_C$  (Henson & Reeves 1999*a,b*), and  $\mathcal{V}$  (Brien & Martin 2000) (a successor of the  $\mathcal{W}$  logic that appeared in early working drafts of Standard Z) fully. It is desirable to support both  $Z_C$  and  $\mathcal{V}$  (and any future proposals for Z logics), since they are rather different and it would be interesting to compare them in a common framework. For example, substitution is a meta-level operation in  $Z_C$  but is defined within the object logic of  $\mathcal{V}$ . To support richer logics such as these, it would be necessary to generalise premisses into sequents that can specify a different context to that of the conclusion. For example, a premiss such as  $x : T \vdash P$  would allow the proof of  $P$  to proceed in an extended context where the new variable  $x$  is declared and has type  $T$ . This is similar to the common sequent calculus style of proof (Girard et al. 1990).

The current CZT reasoning tools construct an explicit proof tree that allows proofs to be recorded, replayed, displayed, checked by independent tools etc. It would be interesting to build a tactic layer on top of this proof tree, using a tactic language such as ANGEL (Martin et al. 1996), which has been used as the basis of several other Z provers such as CADiZ (Toyn 1998), Jigsaw1, Jigsaw2 and Ergo (Martin et al. 1997). This would make it easier to program combinations of rules in flexible ways, whereas currently such tactics must be written as Java methods. If ZedRules was extended to support richer logics and the CZT tools were extended to support tactics, they could be used as practical Z theorem provers (after a large amount of theory development).

In Section 3.4 we defined what it means for an *instantiation* of a rule to be correct. This provides a sufficient basis for the writers of rules to informally check the correctness of the rules that they write. In the future, we would also like to be able to prove the correctness of *uninstantiated* rules, with jokers and provisos. This cannot be done within Z itself, because many provisos require reasoning about the syntax of Z, but it could be done by translating the rules into a *deep embedding* of Z within another prover or logical framework, such as Isabelle or Twelf (Kraan & Baumann 1995, Dawson & Goré 2001). This would be useful for proving correctness of rules, but would probably be impractical as a Z transformation engine, because reasoning using deep embeddings is much more cumbersome than reasoning directly in an object language, and because the rules would have to be expressed within the deep embedding syntax rather than directly in Z syntax. One of the main advantages of our approach is that rules are written directly in an extension of Z.

## References

Armour, P. G. (2000), ‘The case for a new business model’, *Communications of the ACM* **43**(8), 19–22.

Arthan, R. (2007), ‘ProofPower’, on-line documentation.  
**URL:** <http://www.lemma-one.com/ProofPower/>

Brien, S. M. & Martin, A. P. (2000), ‘A calculus for schemas in Z’, *Journal of Symbolic Computation* **30**, 63–91.

Community Z Tools (2008), on-line documentation.  
**URL:** <http://czt.sourceforge.net/>

Dawson, J. & Goré, R. (2001), Embedding display calculi into logical frameworks: Comparing twelf and isabelle, in ‘Computing: The Australasian Theory Symposium (CATS 2001)’, Vol. 42 of *Electronic Notes in Theoretical Computer Science*, Elsevier B.V., pp. 89–103.

Girard, J.-Y., Taylor, P. & Lafont, Y. (1990), *Proofs and Types*, Cambridge University Press, New York, NY, USA.

Hayes, I. (1987), *Specification Case Studies*, Prentice-Hall.

Henson, M. C. & Reeves, S. (1999*a*), ‘Revising Z: Part I - logic and semantics’, *Formal Aspects of Computing* **11**(4), 359–380.

Henson, M. C. & Reeves, S. (1999*b*), ‘Revising Z: Part II - logic development’, *Formal Aspects of Computing* **11**(4), 381–401.

ISO/IEC 13568 (2002), *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics*, ISO/IEC. First Edition 2002-07-01.

Kraan, I. & Baumann, P. (1995), Implementing z in isabelle, in ‘ZUM’95: The Z Formal Specification Notation’, Vol. 976 of *LNCS*, Springer Berlin, pp. 354–373.

Malik, P. & Utting, M. (2005), CZT: A framework for Z tools, in H. Treharne, S. King, M. C. Henson & S. A. Schneider, eds, ‘ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings’, Vol. 3455 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, pp. 65–84.

Martin, A., Nickson, R. & Utting, M. (1997), A tactic language for Ergo, in L. Groves & S. Reeves, eds, ‘Formal Methods Pacific’97’, Springer Series in Discrete Mathematics and Theoretical Computer Science, pp. 186–207.

Martin, A. P., Gardiner, P. H. B. & Woodcock, J. C. P. (1996), ‘A tactic calculus — abridged version’, *Formal Aspects of Computing* **8**(4), 479–489.

ORA Canada (1999), Z/EVES version 1.5: An overview, in D. Hutter, W. Stephan, P. Traverso & M. Ullmann, eds, ‘Applied Formal Methods — FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method’, Vol. 1641 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 367–376.

Spivey, J. M. (1992), *The Z Notation: A Reference Manual*, Prentice Hall International (UK) Ltd., Hertfordshire, UK.

Toyn, I. (1998), A tactic language for reasoning about Z specifications, in ‘3rd BCS-FACS Northern Formal Methods Workshop’.

Toyn, I. (2007), ‘CADiZ’, on-line documentation.  
**URL:** <http://www.cs.york.ac.uk/hise/cadiz/>

van den Brand, M. G. J., Klint, P. & Vinju, J. J. (2003), ‘Term rewriting with traversal functions’, *ACM Transactions on Software Engineering Methodology* **12**(2), 152–190.

Woodcock, J. C. P. & Davies, J. (1996), *Using Z: Specification, Proof and Refinement*, Prentice Hall International Series in Computer Science.  
**URL:** <http://www.usingz.com/>