Transformation Rules for Z

Mark Utting* Petra Malik† Ian Toyn

January 7, 2010

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 [23, 30, 15] 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 [12, 5, 24, 7].

For illustration purposes, consider the Z specification of a birthday book derived from [23]—we go through this example in some detail to introduce Z, to point out a few ways in which ISO Standard Z differs from the earlier Z notation, and to illustrate some of the Z transformations that our rules support.

section *birthdaybook* **parents** *standard_toolkit*

This defines a new Z section called *birthdaybook*. By declaring *standard_toolkit* as a parent, definitions for symbols like $\nrightarrow$ (partial function) are included for later use. The toolkit definitions have been standardised in [15], henceforth referred to as Standard Z.

Z specifications consist of a mixture of narrative text (like this paragraph) and formal Z paragraphs. The following formal 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 partial function from names to dates:

*Department of Computer Science, The University of Waikato, New Zealand
†School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$

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

Init

$$BirthdayBook$$
$$known = \varnothing$$

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 has the effect of declaring the variables in the declaration part of the schema and constraining them as given in the predicate part of the schema. That is, schema *Init* could have been written as:

InitExpanded

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$
$$known = \varnothing$$

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

$$\vdash ?\ Init = InitExpanded$$

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

Next, consider an operation 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 to the given date is added to the existing birthday book entries:

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

The declaration $\Delta BirthdayBook$ is equivalent to a schema that includes both *BirthdayBook* and its decorated version *BirthdayBook* $'$ (note that Standard Z requires a visible space between schema expressions and decorations):

$$\vdash? \ \Delta BirthdayBook = [BirthdayBook;\ BirthdayBook\ ']$$

where $BirthdayBook\ '$ is schema $BirthdayBook$ with its components primed:

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

Using these two equalities, we can give an expanded equivalent definition of *AddBirthday*:

AddBirthdayExpanded

$known : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$
$known' : \mathbb{P}\, NAME$
$birthday' : NAME \pfun DATE$
$name? : NAME$
$date? : DATE$

$known = \mathrm{dom}\, birthday$
$known' = \mathrm{dom}\, birthday'$
$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

$$\vdash? \ AddBirthday = AddBirthdayExpanded$$

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. Other examples of common transformations performed by Z tools include calculating the preconditions of schemas, simplifying schema expressions, and calculating domain checks for predicates [22]. However, in most Z tools, such 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 [2].

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 *Init* and *AddBirthday* 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 [1, 17].

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 the application of rules by three example reasoning engines provided by CZT. 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 [26] 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 [18], 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 [22] 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 [3] 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 premises, 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 sound-

ness. 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 it is shown 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 [6], provides a first illustration of the ZedRules notation:

section *example*

★*Expr* E, F, Y

rule Funct

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

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 $(\_\backslash\_)$ 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 usually cannot be expressed within the Z notation. Oracle paragraphs are used to declare proviso notations as having meta-level interpretations. The ZedRules implementation associates oracles 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 premises, written above the line. The syntax of consequents and premises 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*. An implementation of a parser can then receive distinct tokens for jokers from those of ordinary Z names, easing the parsing of rules. 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 premises that are provisos from other premises, 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. An oracle paragraph should be accompanied by a description that clarifies its 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 premise 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.

section *oracle_example* **parents** *standard_toolkit*, *jokers*

relation $(\_ \setminus \_)$

oracle NotFreeIn
$N \setminus E$

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

Since Z does not provide any object level syntax for types, expression notation is used instead. For example, a typecheck oracle can be defined as follows:

relation $(\_ :: \_)$

$$[E1, E2] \Rrule \_ :: \_ : E1 \leftrightarrow E2$$

oracle TypecheckOracle
$E :: T$

The meta-level interpretation of the notation $E :: T$ is that T is the carrier set of E. Note that T is an expression joker, yet stands for a type. The type of a schema is written as $\mathbb{P}[D \mid true]$ where the schema's signature is written as the declarations D.

The LaTeX mark-ups of rule, oracle and joker paragraphs are distinct from those of Standard Z paragraphs, and hence the new paragraphs are treated as narrative text by other Z tools. Moreover, the scopes of joker declarations exclude ordinary Z paragraphs. This allows us to embed rules within Z specifications, while retaining backwards compatibility with Z tools that do not use the ZedRules notation.

3.2 Type System

The type system ensures that names refer to particular declarations, and that these are combined to form expressions and predicates in conformance with type rules. We first consider informally what we expect the type system to do with rules and oracles, and then give a formal extension to the type inference rules of the ISO Z Standard to specify the desired effects.

Rules and oracles contain names that refer to a new kind of declaration: that of joker declarations. As presented in this paper so far, and in our current implementation, joker declarations appear in separate paragraphs and are in scope only for subsequent rule and oracle paragraphs. An alternative would be to declare jokers locally to an individual rule or oracle. The choice of global scope has the advantage of brevity (which is convenient for this paper), but has disadvantages in practice. With global scope, not only does the name become unusable as an ordinary Z name throughout the scope of the joker, but also renaming a joker is trickier: it can be difficult to be sure that all references to the joker have been renamed, and that the new name for the joker does not capture references already in the specification. For example, consider the following rule:

rule Finset

$$\mathbb{F}_1\, E = \mathbb{F}\, E \setminus \{\varnothing\}$$

It gives an equivalence between $\mathbb{F}_1$ (the set of all non-empty finite subsets of a set E) and $\mathbb{F}$ (the set of all finite subsets of E). This rule would have a completely different meaning if $\mathbb{F}$ or $\mathbb{F}_1$ had been declared as global jokers.

Regardless of whether jokers have global or local scope, the use of distinct tokens for jokers in parsing means that their scopes are enforced by the parser and so do not need to be checked by the type system. It also prohibits local Z declarations of the same name within a rule or oracle paragraph; in other words, there cannot be holes in the scope of a joker declaration. Other names can still refer to Z declarations that are locally quantified within a rule or oracle. Where there is no local or joker declaration of a name, the name can refer to a global declaration of the specification, such as the reference to $\mathbb{F}$ in the Finset rule above, and we say that the name is free in the rule or oracle.

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 *Funct* above to rewrite the equality in the following conjecture.

section *scope_example* **parents** *example*, *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.

The above can be expressed more formally by the following revisions to the Z Standard. These revisions are expressed using the metalanguages defined in Clause 4 of [15].

- Add a new metavariable for jokers, J, with the definition "denotes a Joker phrase." in the meta-language defined in Table 20 on page 10.

- Add a new type inference rule for oracle paragraphs, in 13.2.4 as follows.

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \mathbf{oracle}\ i\ p}$$

- Add a new type inference rule for rule paragraphs, in 13.2.4 as follows.

$$\frac{\Sigma \vdash^{\mathcal{P}} p_1 \quad \ldots \quad \Sigma \vdash^{\mathcal{P}} p_n \qquad \Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \begin{array}{l} \mathbf{rule}\ i \\ p_1 \\ \vdots \\ p_n \\ \hline p \end{array}}$$

- Add a new type inference rule for expression jokers, in 13.2.6 as follows.

  $$\frac{}{\Sigma \vdash^{\mathcal{E}} J \mathrel{:\!\circ} \tau}$$

  This places a type annotation on the expression joker, as constrained by the context in which the expression appears. This type is used in constraining instantiations of the rule containing the expression (see below).

- Add a new type inference rule for each other syntactic category of joker, in 13.2, such as the following for predicates.

  $$\frac{}{\Sigma \vdash^{\mathcal{P}} J}$$

  These rules always succeed; they are offered only so that there are rules for the new notation.

The type annotations inferred for expression jokers can be used in typechecking instantiations of rules. In the case of the example rule *Funct* above, the expression jokers have type annotations as follows: $E \mathrel{:\!\circ} \tau_1$, $Y \mathrel{:\!\circ} \tau_2$ and $F \mathrel{:\!\circ} \mathbb{P}(\tau_1 \times \tau_2)$, where τ_1 and τ_2 are meta-variables denoting types. In the case of the example application above of rule *Funct*, the respective instantiations of the jokers have type annotations as follows: $2 \mathrel{:\!\circ} \mathbb{A}$, $n \mathrel{:\!\circ} \mathbb{A}$ and $succ \mathrel{:\!\circ} \mathbb{P}(\mathbb{A} \times \mathbb{A})$. A rule instantiation is type correct if jokers' type annotations can be unified with the types of their instantiations. In the example, there is a unifier (τ_1 is $\mathbb{A}$ and τ_2 is $\mathbb{A}$) and so the rule instantiation is type correct.

An alternative way of typechecking a rule instantiation would be to instantiate the rule and then typecheck the resulting instantiated rule. There would then be no need to typecheck uninstantiated rules, and hence fewer additions to the type inference rules would be necessary, but this would delay

detection of mistakes that are independent of instantiations, and would probably be less efficient overall.

A further issue concerns free names. If a rule is written in a different Z section from a paragraph to which the rule is applied, then the environments of global 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.

3.3 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 subterms 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).

Informally, a rule is valid if it cannot be used to derive a false conclusion from true premises when all of its provisos are true. Using meta-notation based on that of the ISO Z standard, a rule without jokers, with non-proviso premises $p_1, ..., p_n$, and with consequent p is valid if all of its provisos are true and

$$[\![\ p_1 \land \ldots \land p_n \Rightarrow p\ ]\!]^{\mathcal{P}} \supseteq [\![\ z\ ]\!]^{\mathcal{Z}}\ i$$

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 premises 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 rules with provisos that are not true. Note that any object-level constraints on proviso notations have no effect on rule semantics.

Rules that contain jokers are schemata in which the jokers can be instantiated to terms to give well-typed instances of the rule. A rule containing jokers is valid if all instances of the rule are valid. 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 premises 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 premises.

4 Implementation of ZedRules

Several tools have been implemented in the CZT system to use and test ZedRules. This includes a rewrite engine, an automated depth-first prover, and an interactive prover. Note that none of these is intended to be used as a general Z theorem prover. 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 for heavy-duty reasoning with lots of rules. Furthermore, for practical theorem proving, it would be 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.

The rest of this section first discusses implementation issues related to all of the tools: matching and unification in Section 4.1 and provisos in Section 4.2 and then continues by discussing each of the tools separately: a rewrite engine in Section 4.3, an automated theorem prover in Section 4.4 and an interactive prover in Section 4.5.

4.1 Matching and Unification

In order to determine whether a rule can be applied to a term, matching and unification of Z terms at the abstract syntax tree level is used. Matching and unification have the objective of determining whether two terms can be made equal by means of instantiation of jokers. While matching is concerned with the case that only one of the terms contains jokers, unification handles the case where jokers are allowed in both terms. One difficulty that arises in unification is the possibility of creating cyclic structures by instantiating a joker with a term that contains that joker. The CZT implementation of ZedRules does an occurs-check to detect and avoid this case.

Matching and unification are based on an equality notion of terms. But when are two Z terms equal? Which of the following terms are equal?

1. $\forall x : \mathbb{N} \bullet x < 0$
2. $\forall x : \mathbb{N} \bullet \quad x < 0$
3. $\forall y : \mathbb{N} \bullet y < 0$
4. $\forall x : \mathbb{N} \bullet 0 > x$
5. $\forall x : \mathbb{N} \mid true \bullet x < 0$
6. $\forall x : \mathbb{N} \bullet (x, 0) \in \_ < \_$

The equality notion used within CZT is based on the abstract syntax tree (AST) for Z [17], which in turn is based on ZML [27], an XML markup for Z. Each AST node is of a particular type and has a list of child nodes. Two AST nodes are considered equal if they are of the same type and have equal children.

This means that the term equality used in the current CZT implementation is closely related to how the AST is defined, and might be surprising for someone unfamiliar with the CZT AST. Some syntactically different but semantically equivalent terms like $1 < 2$ and $(1, 2) \in \_ < \_$ are considered equal in CZT while the terms $[x : \mathbb{N} \mid true]$ and $[x : \mathbb{N}]$ are considered unequal in the CZT AST.

The CZT AST has 'scope IDs' on names, as set and used by the typechecker. The names of different declarations have different scope IDs, even if their names are spelt the same, unless the declarations are merged within the same schema text. The name in a reference has the same scope ID as the name of the referenced declaration. Two names are considered equal iff they have the same spelling and the same scope ID. This means that none of the terms in the above list are equal to each other, since each declaration within the quantifiers is assigned a distinct scope ID. To overcome this limitation, name jokers can be used. Terms 1, 2, 3, 5, and 6 all match $\forall\, v : \mathbb{N} \bullet v < 0$ where v is a name joker.

Rewrite rules that rearrange terms so that references are moved into different scopes may give rise to the variable capture problem. In the following rule, references in $E1$ may become captured by declarations in D.

rule comprehension

$$E1 \in \{D \bullet E2\} \Leftrightarrow \exists\, D \bullet E1 = E2$$

By maintaining scope IDs on names through an application of a rewrite rule, references are not captured by different declarations. On printing the result as Z, the scope IDs are lost and some renaming may be necessary.

Another problem arises with list jokers. Consider two declaration list jokers $D1$ and $D2$; do the terms

$$\forall\, x : \mathbb{N};\ y : \mathbb{N} \bullet \mathit{true}$$

and

$$\forall\, D1;\ v : \mathbb{N};\ D2 \bullet \mathit{true}$$

match? There are actually two possibilities to make these two terms equal. One is to instantiate $D1$ with the empty list, v with the name x, and $D2$ with $y : \mathbb{N}$. The second possibility is to instantiate $D1$ with $x : \mathbb{N}$, v with the name y, and $D2$ with the empty list. CZT currently does not allow patterns for which several solutions might exist: only one list joker is allowed within a list and it must be used at the end of the list. Thus the above pattern is not accepted, while

$$\forall\, v : \mathbb{N};\ D2 \bullet \mathit{true}$$

is accepted as a valid pattern.

4.2 Provisos

Provisos express useful meta-level properties of the terms in a rule, such as properties of signatures, and distinguishing different decorations on names. In the ZedRules implementation, each oracle is associated to code that checks the corresponding meta-level property. The evaluation of a proviso takes into account any known instantiations of jokers. If all jokers are instantiated, 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.

For the *TypecheckOracle* given in Section 3.1, 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 associate constraints with 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.

4.3 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 [28]. Sub-terms are rewritten in a left-to-right, bottom-up fashion until no more rewrite rules can be applied. The author of the rules needs to ensure termination of the rules.

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

4.4 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 subgoals is created from the premises 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.

4.5 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.

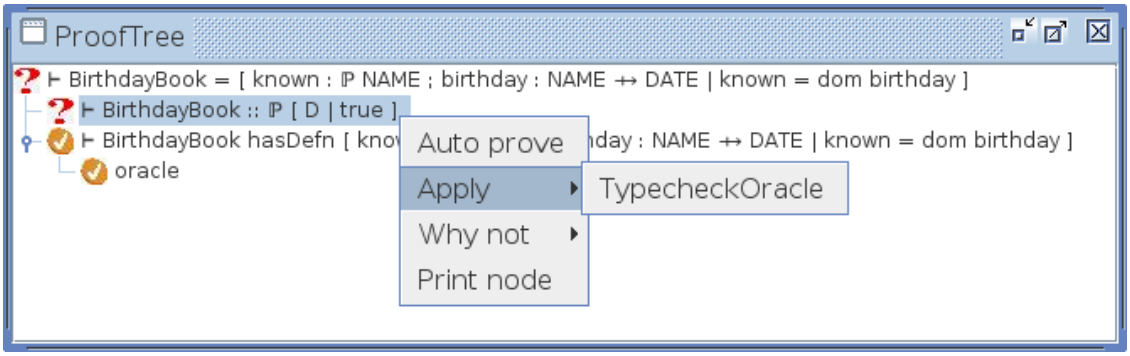

Figure 1: Screenshot of Interactive Prover

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 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 premises are merely matched.

relation (_ :: _)

$$\frac{[E1, E2]}{\_ :: \_ : \mathbb{P}(E1 \times E2)}$$

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 (_ **hasDefn** _)

$$\frac{[E]}{\_ \ \mathbf{hasDefn} \ \_ : \mathbb{P}(E \times E)}$$

oracle LookupOracle

$E1$ **hasDefn** $E2$

$E1$ **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.

relation (_ **is** _)

$$\frac{[E1, E2]}{\_ \ \mathbf{is} \ \_ : \mathbb{P}(E1 \times E2)}$$

oracle DecorOracle

$E \ ^{S}$ **is** $E2$

$E \ ^{S}$ **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

θE **is** $E2$

θE **is** $E2$ is true provided that $E2$ is a binding extension and the binding constructed by θE is the same as $E2$. $E2$ can be calculated deterministically from E, but not E from $E2$.

oracle DecorThetaOracle

$\theta E\ ^{S}$ **is** $E2$

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

oracle SchemaMinusOracle

$[D1 \mid true] \setminus [D2 \mid true]$ **is** $[D3 \mid true]$

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

oracle UnprefixOracle

$N1$ *unprefix* $N2$ **is** $N3$

$N1$ *unprefix* $N2$ **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

$(split[D \mid true])$ **is** $([D1 \mid true]? \land [D2 \mid true] \land [D3 \mid true]' \land [D4 \mid true]!)$

$(split[D \mid true])$ **is** $([D1 \mid true]? \land [D2 \mid true] \land [D3 \mid true]' \land [D4 \mid true]!)$ 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, $(split[x, x', y?, y!, z : T])$ **is** $([y : T]? \land [x : T;\ z : T] \land [x : T]' \land [y : T]!)$.

oracle HideOracle

$[D1 \mid true] \setminus (NL)$ **is** $\exists [D2 \mid true] \bullet [D1 \mid true]$

$[D1 \mid true] \setminus (NL)$ **is** $\exists [D2 \mid true] \bullet [D1 \mid true]$ 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

$E[RL]$ **is** $E2$

$E[RL]$ **is** $E2$ is true provided that the renaming of schema E is the same as schema $E2$.

oracle XiOracle

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

$(\theta[D \mid true] = \theta[D \mid true]\ ') \Leftrightarrow P$ is true provided that P is a conjunction of equalities and the truth of the predicate from a Ξ schema is equivalent to 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 $E \land [x : \mathbb{N} \mid p(x)]$ 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 4.3) 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

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

rule PredVarDecl2

$$pred[N : E;\ D \mid true] \Leftrightarrow N \in E \land pred[D \mid true]$$

rule PredConstDecl

$$pred[N == E;\ D \mid true] \Leftrightarrow N = E \land pred[D \mid true]$$

rule PredIncludeDecl

$$pred[E;\ D \mid true] \Leftrightarrow E \land pred[D \mid true]$$

rule PredEmptyDecl

$$pred[\mid true] \Leftrightarrow true$$

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

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

For example, if E is *BirthdayBook*, then D is calculated to be

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

and then $E2$ is calculated to be

$$\langle\!| known == known, birthday == birthday |\!\rangle.$$

The next rule handles decorated θ 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

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

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

rule SchemaDecor

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

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

rule SchemaRenaming

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

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

rule SchemaNegation

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

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.

rule SchemaConjunction

$$\frac{([D1 \mid true] \land [D2 \mid true]) :: \mathbb{P}[D3 \mid true]}{([D1 \mid P1] \land [D2 \mid P2]) = [D3 \mid pred[D1 \mid true] \land P1 \land pred[D2 \mid true] \land P2]}$$

rule SchemaDisjunction

$$\frac{([D1 \mid true] \land [D2 \mid true]) :: \mathbb{P}[D3 \mid true]}{([D1 \mid P1] \lor [D2 \mid P2]) = [D3 \mid (pred[D1 \mid true] \land P1) \lor (pred[D2 \mid true] \land P2)]}$$

rule SchemaImplication

$$\frac{([D1 \mid true] \land [D2 \mid true]) :: \mathbb{P}[D3 \mid true]}{([D1 \mid P1] \Rightarrow [D2 \mid P2]) = [D3 \mid (pred[D1 \mid true] \land P1) \Rightarrow (pred[D2 \mid true] \land P2)]}$$

rule SchemaEquivalence

$$\frac{([D1 \mid true] \land [D2 \mid true]) :: \mathbb{P}[D3 \mid true]}{([D1 \mid P1] \Leftrightarrow [D2 \mid P2]) = [D3 \mid (pred[D1 \mid true] \land P1) \Leftrightarrow (pred[D2 \mid true] \land P2)]}$$

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.

rule SchemaProjection

$$\frac{\begin{array}{l}([D1 \mid true] \land [D2 \mid true]) :: \mathbb{P}[D3 \mid true] \\ [D2 \mid true] :: \mathbb{P}[D4 \mid true] \\ ([D3 \mid true] \setminus [D4 \mid true]) \textbf{ is } [D5 \mid true]\end{array}}{[D1 \mid P1] \upharpoonright [D2 \mid P2] = [D2 \mid (\exists\, D5 \bullet pred[D1 \mid true] \land P1 \land P2)]}$$

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 $D6$.

rule SchemaPrecondition

$$\frac{\begin{array}{l}[D \mid true] :: \mathbb{P}[D1 \mid true] \\ (split[D1 \mid true]) \textbf{ is } ([D2 \mid true]? \land [D3 \mid true] \land [D4 \mid true]' \land [D5 \mid true]!) \\ ([D2 \mid true]? \land [D3 \mid true]) :: \mathbb{P}[D6 \mid true]\end{array}}{\mathrm{pre}[D \mid P] = [D6 \mid (\exists([D4 \mid true]' \land [D5 \mid true]!) \bullet pred[D \mid true] \land P)]}$$

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

rule SchemaExists

$[D1 \mid true] :: \mathbb{P}[D3 \mid true]$
$[D2 \mid true] :: \mathbb{P}[D4 \mid true]$
$([D4 \mid true] \setminus [D3 \mid true])$ **is** $[D5 \mid true]$

$(\exists\, D1 \mid P1 \bullet [D2 \mid P2]) = [D5 \mid (\exists\, D1 \bullet P1 \wedge pred[D2 \mid true] \wedge P2)]$

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

rule SchemaExists1

$[D1 \mid true] :: \mathbb{P}[D3 \mid true]$
$[D2 \mid true] :: \mathbb{P}[D4 \mid true]$
$([D4 \mid true] \setminus [D3 \mid true])$ **is** $[D5 \mid true]$

$(\exists_1\, D1 \mid P1 \bullet [D2 \mid P2]) = [D5 \mid (\exists_1\, D1 \bullet P1 \wedge pred[D2 \mid true] \wedge P2)]$

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

rule SchemaForall

$[D1 \mid true] :: \mathbb{P}[D3 \mid true]$
$[D2 \mid true] :: \mathbb{P}[D4 \mid true]$
$([D4 \mid true] \setminus [D3 \mid true])$ **is** $[D5 \mid true]$

$(\forall\, D1 \mid P1 \bullet [D2 \mid P2]) = [D5 \mid (\forall\, D1 \bullet P1 \wedge pred[D2 \mid true] \wedge P2)]$

Schema composition is one of the most complex schema operators. Nevertheless, this Schema-Composition 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 appear in $D2$. Note that we currently use the rarely used decoration $_9$. In the future, we intend to consider using a special decoration that cannot be entered by users, rather like $\bowtie$ [15].

rule SchemaComposition

$E1 :: \mathbb{P}[D1 \mid true]$
$E2 :: \mathbb{P}[D2 \mid true]$
$(split[D1 \mid true])$ **is** $([D3 \mid true]? \wedge [D4 \mid true] \wedge [D5 \mid true]' \wedge [D6 \mid true]!)$
$([D5 \mid true] \setminus [D2 \mid true])$ **is** $E3$
$([D5 \mid true] \setminus E3)$ **is** $E4$

$E1 \fatsemi_9 E2 = (\exists\, E4\ _9 \bullet (\exists\, E4\,' \bullet [E1 \mid \theta E4\,' = \theta E4\ _9]) \wedge (\exists\, E4 \bullet [E2 \mid \theta E4 = \theta E4\ _9]))$

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

rule SchemaPiping
$E1 :: \mathbb{P}[D1 \mid true]$
$E2 :: \mathbb{P}[D2 \mid true]$
$(split[D1 \mid true])$ **is** $([D3 \mid true]? \wedge [D4 \mid true] \wedge [D5 \mid true]' \wedge [D6 \mid true]!)$
$(split[D2 \mid true])$ **is** $([D7 \mid true]? \wedge [D8 \mid true] \wedge [D9 \mid true]' \wedge [D10 \mid true]!)$
$([D6 \mid true] \setminus [D7 \mid true])$ **is** $E3$
$([D6 \mid true] \setminus E3)$ **is** $E6$

$E1 \gg E2 = (\exists\, E6_{\ 9} \bullet (\exists\, E6\ ! \bullet [E1 \mid \theta E6\ ! = \theta E6_{\ 9}]) \wedge (\exists\, E6\ ? \bullet [E2 \mid \theta E6\ ? = \theta E6_{\ 9}]))$

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

rule SchemaHiding
$E :: \mathbb{P}[D1 \mid true]$
$[D1 \mid true] \setminus (NL)$ **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
$N[EL] :: \mathbb{P}[D \mid true]$
$N[EL]$ **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
$N :: \mathbb{P}[D \mid true]$
N **hasDefn** $E2$

$N = E2$

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

rule DeltaRef
Δ *unprefix* N **is** $N2$

$N = [N2;\ N2\ ']$

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

rule XiRef
Ξ *unprefix* N **is** $N2$
$[N2 \mid true] :: \mathbb{P}[D2 \mid true]$
$\theta[D2 \mid true] = \theta[D2 \mid true]\ ' \Leftrightarrow P$

$N = [N2;\ N2\ ' \mid P]$

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

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

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 4.3) 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 rules 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

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

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 $\mathbb{N} \to \{0, 1\}$. 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 $\mathbb{N} \to \{0, 1\}$ into the temporary **let** form, which is then translated into a RelSet object that is equivalent to $\mathbb{N} \to \{0, 1\}$.

rule fun

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

rule bij

$$E1 \rightarrowtail\!\!\!\!\!\twoheadrightarrow E2 = (\textbf{let}\ isFun == 1;\ isTotal == 1;\ isOnto == 1;\ isInj == 1 \bullet \mathbb{P}(E1 \times E2))$$

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

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

and this is translated into the RelSet object

$$\{f : \mathbb{P}(\mathbb{N} \times \{0, 1\}) \mid f \in \mathbb{N} \pfun \{0, 1\} \land \operdom f = \mathbb{N}\}$$

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 [15].

The ZedRules notation can be viewed as an approach to making transformations configurable, analogous to Banach's idea of configurable generation of proof obligations for model-based refinement tools [4]. The realisation of his idea in the Frog toolkit [9] uses templates for proof obligations that are instantiated in an analogous way to the instantiation of our rule schemata.

In the future, we would like to be able to extend the ZedRules notation to fully support a variety of Z logics, such as Z_C [13, 14], and $\mathcal{V}$ [6]. $\mathcal{V}$ is a successor of the $\mathcal{W}$ logic [29, 11] that appeared in early working drafts of Standard Z. 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 premises into sequents that can specify a different context to that of the conclusion. For example, a premise 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 [10].

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 [18] or ArcAngel [20]. Such tactic languages have successfully been used in other Z provers such as CADiZ [25], Jigsaw1, Jigsaw2 and Ergo [19], and refinement tools such as Refine and Gabriel [21]. A tactic layer 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 check informally 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 [16, 8]. 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

[1] Community Z Tools. on-line documentation, 2009.

[2] Philip G. Armour. The case for a new business model. *Communications of the ACM*, 43(8):19–22, August 2000.

[3] Rob Arthan. ProofPower. on-line documentation, 2009.

[4] Richard Banach. Model based refinement and the tools of tomorrow. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 42–56, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach.* International Thomson Computer Press, 1996.

[6] S. M. Brien and A. P. Martin. A calculus for schemas in Z. *Journal of Symbolic Computation*, 30:63–91, 2000.

[7] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. Technical report, W3C, 2007. Published W3C Recommendation, 26 June 2007.

[8] Jeremy Dawson and Rajeev Goré. Embedding display calculi into logical frameworks: Comparing Twelf and Isabelle. In *Computing: The Australasian Theory Symposium (CATS 2001)*, volume 42 of *Electronic Notes in Theoretical Computer Science*, pages 89–103. Elsevier B.V., 2001.

[9] Simon Fraser and Richard Banach. Configurable proof obligations in the frog toolkit. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10–14 September 2007, London*, pages 361–370. IEEE Computer Society, 2007.

[10] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, New York, NY, USA, 1990.

[11] Jon Hall and Andrew Martin. W reconstructed. In Jonathan P. Bowen, Michael G Hinchey, and David Till, editors, *ZUM'97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 1997, Proceedings*, Berlin, Heidelberg, April 1997. Springer-Verlag.

[12] I. J. Hayes. *Specification Case Studies*. Prentice Hall International, second edition, 1993.

[13] Martin C. Henson and Steve Reeves. Revising Z: Part I - logic and semantics. *Formal Aspects of Computing*, 11(4):359–380, 1999.

[14] Martin C. Henson and Steve Reeves. Revising Z: Part II - logic development. *Formal Aspects of Computing*, 11(4):381–401, 1999.

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

[16] Ina Kraan and Peter Baumann. Implementing Z in Isabelle. In *ZUM'95: The Z Formal Specification Notation*, volume 976 of *LNCS*, pages 354–373. Springer-Verlag, 1995.

[17] Petra Malik and Mark Utting. CZT: A framework for Z tools. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *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*, volume 3455 of *Lecture Notes in Computer Science*, pages 65–84, Berlin, Heidelberg, 2005. Springer-Verlag.

[18] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A tactic calculus — abridged version. *Formal Aspects of Computing*, 8(4):479–489, July 1996.

[19] Andrew Martin, Ray Nickson, and Mark Utting. A tactic language for Ergo. In L. Groves and S. Reeves, editors, *Formal Methods Pacific'97*, Springer Series in Discrete Mathematics and Theoretical Computer Science, pages 186–207, 1997.

[20] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a tactic language for refinement. *Formal Aspects of Computing*, 15(1):28–47, 2003.

[21] M. V. M. Oliveira, M. Xavier, and A. L. C. Cavalcanti. Refine and Gabriel: Support for refinement and tactics. In Jorge R. Cuellar and Zhiming Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310 – 319. IEEE Computer Society Press, September 2004.

[22] ORA Canada. Z/EVES version 1.5: An overview. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Applied Formal Methods — FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, volume 1641 of *Lecture Notes in Computer Science*, pages 367–376, London, UK, 1999. Springer-Verlag.

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

[24] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[25] Ian Toyn. A tactic language for reasoning about Z specifications. In *3rd BCS-FACS Northern Formal Methods Workshop*, 1998.

[26] Ian Toyn. CADiZ. on-line documentation, 2006.

[27] Mark Utting, Ian Toyn, Jing Sun, Andrew Martin, Jin Song Dong, Nicholas Daley, and David W. Currie. ZML: XML support for Standard Z. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, pages 437–456. Springer-Verlag, 2003.

[28] Mark G. J. van den Brand, Paul Klint, and Jurgen J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering Methodology*, 12(2):152–190, 2003.

[29] J. C. P. Woodcock and S. M. Brien. W: A logic for Z. In J. E. Nicholls, editor, *Z User Workshop, York, UK, 16-17 December 1991, Proceedings*, Workshops in Computing. Springer-Verlag, 1992.

[30] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.