Logical Frameworks as a Basis for Verification Tools: A Case Study

Ina Kraan and Peter Baumann

inak,baumann@ifi.unizh.ch
Institut für Informatik der Universität Zürich
Winterthurerstrasse 190, 8057 Zürich, Schweiz

May 12, 1995

Abstract

Wide-spread acceptance and use of formal methods in software development hinges on the availability of powerful tools. Tools must be both reliable and offer real assistance to the user. Logical frameworks are a suitable medium to build such tools, since they provide a means to show the faithfulness and adequacy of the implementation, and at the same time provide the flexibility needed to build sufficiently automated tools. We present Z-in-Isabelle, a deep semantic embedding of the specification language Z and a deductive system for Z in the generic theorem prover Isabelle. Z is based on Zermelo-Fraenkel set theory and first-order predicate logic, extended by a notion of schemas. Isabelle supports a fragment of higher-order predicate logic, in which object logics such as Z can be encoded as theories. We illustrate the use of Z-in-Isabelle with a data refinement proof. We assess to what extent such proofs need to and can be automated to make implementations in logical frameworks such as Z-in-Isabelle viable tools for reasoning about specifications.
Keywords: Verification, proof tools, formal methods, logical frameworks.

1 Introduction

Wide-spread acceptance and use of formal methods in software development depends on the availability of suitable tools. Since applying formal methods usually involves some kind of proof, tools for formal methods are primarily proof tools. To be useful, they must therefore both be reliable and offer real assistance to the user. We believe that logical frameworks are a suitable medium to build such tools, since they provide a means to show the faithfulness and adequacy of the implementation, and at the same time provide the flexibility needed to build intelligent tools which automate large parts of the proofs involved in the application of formal methods, which tend to be large and to involve many repetitive and tedious steps.

We have implemented the specification language Z and a deductive system for Z in the generic theorem proving tool Isabelle. The implementation is based on Z and its deductive system as defined in the draft Z standard [Brien & Nicholls 92]. It is a deep embedding of Z, i.e., both the syntax and a deductive system are formalized within Isabelle's meta-logic.

Implementations of formal languages can be roughly divided into two categories. First, there are the direct implementations. For Z, these include the tool Balzac [Harwood 91, Jordan 91]. Second, there are the embeddings in other logics and in logical frameworks. Most such implementations of Z have been shallow rather than deep embeddings, including ProofPower [Jones 92], Z in HOL [Bowen & Gordon 94], and Z in LEGO [Maharaj 90]. The distinction between shallow and deep is not always clear-cut, but in principle, the deeper the embedding, the more properties of the language can be proven in the embedding. Deep embeddings of Z in logical frameworks include Jigsaw [Martin 94] and our system Z-in-Isabelle. Despite being deep embeddings within logical frameworks, the two systems are quite different (see section 6).

An embedding within a logical framework has the advantage that one can prove the faithfulness and adequacy of the implementation more easily (for an example of such proofs for first-order logic in Isabelle, see [Paulson 89]). A deep embedding, while involving much more effort, has a number of advantages over shallower ones. In particular, it can be used to rigorously derive, as theorems in the meta-logic, new rules of inference for the object logic. These may be more desirable because they are simpler to use, or because they facilitate the automation of proofs. In fact, the possibility of deriving rules of inference has proved invaluable in Z-in-Isabelle, not least from the point of view of efficiency and automation.

In this paper, we present Z-in-Isabelle and a case study of a verification proof in Z-in-Isabelle. Using this case study, we demonstrate that a deep semantic embedding of Z can indeed form the basis of a reliable and powerful tool to automate reasoning about Z specifications.

The paper assumes little knowledge of Isabelle or Z. It does, however, assume knowledge of the formal systems underlying Isabelle and Z, i.e., first-order and higher-order logic, the λ-calculus and Zermelo-Fraenkel set theory. The paper starts with a brief introduction to Isabelle and to Z. We then present an overview of Z-in-Isabelle, explaining some of the crucial design decisions. Subsequently, we present an example proof of a proof obligation in data refinement within Z, discussing to what extent such proofs can and need to be automated to make Z-in-Isabelle a useful tool. Finally, we present our conclusions and point out directions for future work.

2 The Generic Theorem Prover Isabelle

Isabelle [Paulson 94] is a generic theorem prover. Its distinctive feature is that it represents logics in a fragment of higher-order logic. The implemented (object) logic, here Z, is implemented as a theory in Isabelle's (meta-)logic, where each inference rule of the object logic is an axiom in the meta-logic. Isabelle provides a powerful mechanism to define theories, which allows a flexible translation between concrete and abstract syntax. Furthermore, Isabelle provides a tactic[1] mechanism, which can be used to automate proofs.

Isabelle represents syntax using the simply typed λ-calculus. It provides polymorphic types, and uses type classes to control polymorphism. Variable binding is expressed using λ-abstraction. This technique is also known as higher-order abstract syntax [Pfenning & Elliot 88]. For example, $\forall\, x.\ P$ is represented as $\forall(\lambda\, x.\ P)$. Also, $F(t)$ represents the application of the function F to the argument t. If F has the form $\lambda\, x.\ P$, Isabelle will β-reduce $(\lambda\, x.\ P)(t)$ to $P[t/x]$. This enables us to express inference rules that involve substitution for bound variables in such a way that the usual side conditions (e.g., "x is not free in ...") are respected. Thus, for instance,

[1]Tactics are programs that apply inference rules.

universal introduction can be formalized in Isabelle as

$$(\bigwedge x.\ Trueprop(P(x))) \Longrightarrow Trueprop(\forall(\lambda x.P(x)))\ .$$

The function *Trueprop* coerces object-level truth to meta-level truth, and $\bigwedge$ is meta-level universal quantification. Thus, we can read the above as: "if $P(x)$ is true for all x, then $\forall x.\ P(x)$ is true". Fortunately, Isabelle hides much of this meta-logic "overhead". The above rule would actually be entered as something like[2]:

```
(!!x. P(x)) ==> ALL x. P(x)
```

Isabelle allows schematic variables or unknowns, which can be instantiated via unification. Schematic variables are distinguished from other variables by a leading question mark. This is particularly useful when applying inferences that introduce new parameters or variables. For instance, existential introduction can be formalized as

```
P(?x) ==> EX x. P(x)
```

This rule introduces a new schematic variable, `?x`, representing the witness for the existential variable `x`. Thus, the rule can be applied without having to provide the actual witness. The schematic variable will become instantiated to the witness via unification later in the proof.

Isabelle's meta-logic is a fragment of higher-order logic, with universal quantification, implication, and equality. The inference rules of the logic are almost never used directly in proofs, however. Normally, a variety of derived resolution rules are used. These rules are tailored in such a way that a resolution step in Isabelle's meta-logic mimics the application of a rule in the object logic. Thus, the use of a meta-logic is transparent to quite a large extent. Isabelle allows both forward-chaining proof, i.e., two theorems can be resolved against each other to produce a new theorem, and backward-chaining proof, i.e., a conjecture (goal) is resolved against a theorem, which yields one or more new conjectures (subgoals). In backward proof, the original conjecture is a theorem when all subgoals have been reduced to *true* within a given theory. The proof shown in this paper is a backward proof.

3 The Specification Language Z

The specification language Z [Brien & Nicholls 92] is based on first-order predicate logic and Zermelo-Fraenkel set theory, extended with a notion of schemas. Schemas are in fact the main construct of the language. A schema declares a set of variables and constraints. The meaning of a schema is the set of bindings (substitutions) for the variables that satisfy the constraints. Schemas can also be used as declarations, expressions and predicates. When reasoning about Z specifications, schemas are most frequently used as predicates.

Schemas can be combined flexibly. The following schema, for instance, declares an abstract state of a system that manages a database of names and birthdays (the example is taken from [Spivey 92]):

BirthdayBook

$known : \mathbb{P}\, NAME$
$birthday : NAME \rightarrow\!\!\!\!\!+ DATE$

$known = \mathrm{dom}\ birthday$

[2] The symbol `!!` stands for meta-level universal quantification ($\bigwedge$), and `==>` for meta-level implication ($\Longrightarrow$).

The variable *known* is a set of names, *birthday* is a partial function from names to dates, and *known* equals the domain of *birthday*. Using this abstract state, we can specify the operation of adding a name-birthday pair.

AddBirthday

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

AddBirthday imports *BirthdayBook*, using the Δ convention. This means that *AddBirthday* includes two copies of *BirthdayBook*, one of whose variables are decorated with dashes. Undashed variables are those of the state before the operation, dashed variables those of the state after the operation. Variables decorated with questions marks are input variables and variables with exclamation points output variables. *AddBirthday* specifies that the name to be added may not already be in the set of names in the database, and that the new database is the union of the old database and the new name-birthday pair.

The core language of Z is extended by the *mathematical toolkit*, which contains a large number of definitions that are generally useful. The definitions of $\nrightarrow$, dom, $\notin$, $\cup$, and $\mapsto$ used above, for instance, are among those in the toolkit.

Z has been designed as a powerful specification language that enables writing large specifications in a concise manner. Mechanization was not an uppermost priority, and, not surprisingly, some of the design decisions taken in the language to ease the writing of specifications have made implementing and reasoning within the language difficult. The major design decisions and issues are discussed in the following section, which provides an overview of the system.

4 Z-in-Isabelle

4.1 Overview

Z has been implemented as a family of theories in Isabelle (see Figure 1). The theory *Pure* contains Isabelle's meta-logic. The basic theory underlying Z is *Lkz*. *Lkz* defines sequents using the representation of the standard LK theory distributed with Isabelle. It also defines the type system of Z-in-Isabelle, which mirrors the type system of Z to a large extent, but is less detailed (see Figure 2). There are two type constructors, *Set* (a postfix operator) and $*$ (an infix operator), that construct objects of type *Power* and *Cartesian*, respectively, from objects of type *Type*. Theories for the given (built-in) types integers (*Number*) and strings (*String*) extend *Lkz*. The theory *Z* is based on these three theories, and defines the core language and the deductive system of Z. Further theories define substitution and the mathematical toolkit.

A specification is implemented as a theory extending the family of theories in Figure 1. An Isabelle theory file consists of a signature, which contains the information necessary for type checking, parsing, and pretty printing terms, and of a set of axioms. We illustrate again using the BirthdayBook example from [Spivey 92]. Here is part of the BirthdayBook theory in Isabelle (BB.thy):

```
BB = Toolkit +

types
```

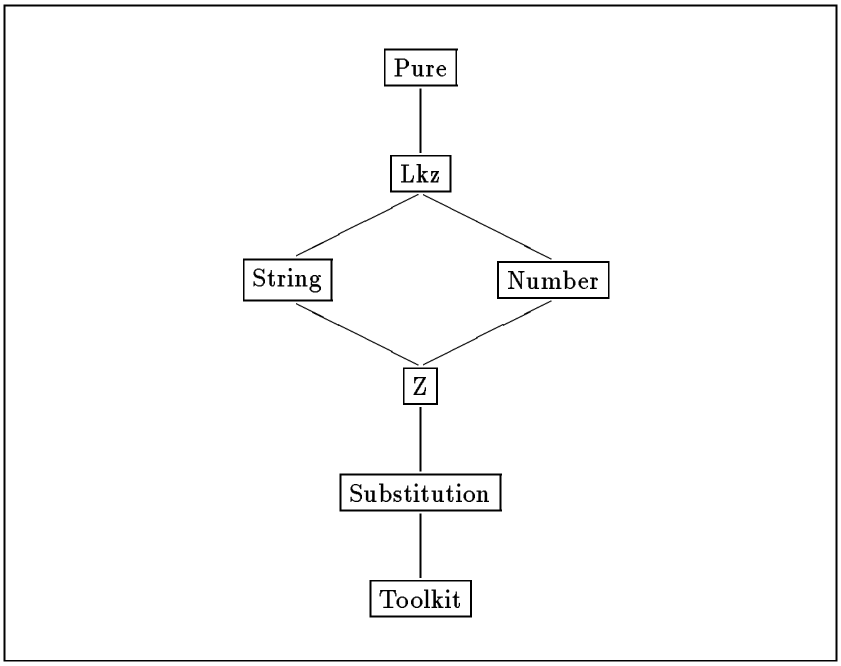

Figure 1: Overview of Z-in-Isabelle theories.

```
  NameType, DateType 0

arities
  NameType, DateType :: Given

consts
  NAME                          :: "NameType Set"
  DATE                          :: "DateType Set"
  BirthdayBook                  :: "Schema"
  known                         :: "NameType Set"
  birthday                      :: "( NameType * DateType ) Set"

translations
   "BirthdayBook" == "[ known : \power NAME;
                        birthday : NAME \pfun DATE
                      |
                        known = \dom birthday
                      ]"
end
```

The first part defines the Isabelle types and their arities. The types *NameType* and *DateType* belong to the class *Given*, and they take no parameters. The second part defines the constants and their Isabelle types. *NAME*, for instance, is of Isabelle-type *NameType Set*, which means that it is a set of elements belonging to the given type *NameType*. The last part contains the translation between the concrete and the abstract syntax of the *BirthdayBook* schema (see section 4.2 for details). Definitions other than schemas, e.g., axiomatic definitions and global constraints, correspond to

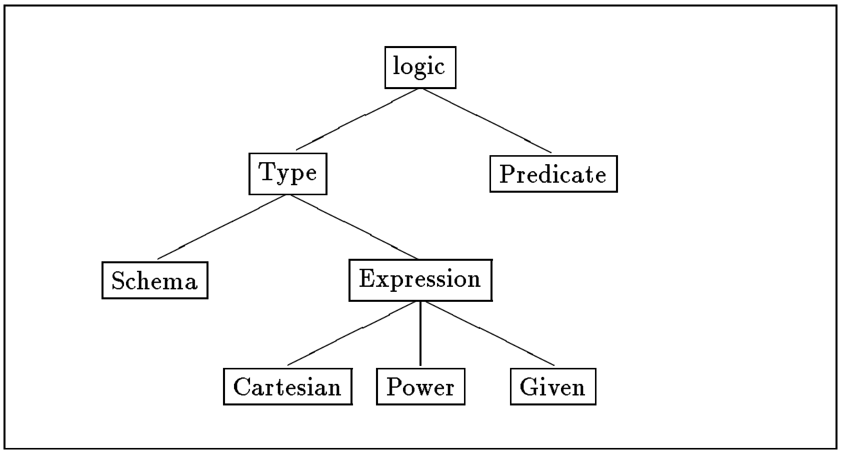

Figure 2: Overview of Z-in-Isabelle classes.

axioms and would be declared in a further section called `rules`.

Though we have tried to keep the syntax of Z-in-Isabelle close to the standard LaTeX style (in which Z specifications are often written), a number of syntactic requirements on the format of theory files and a few lexical quirks of Isabelle make a certain amount of preprocessing inevitable. A preprocessor which type-checks specifications and translates them into Z-in-Isabelle format is currently under development.

The inference system we have implemented is that in Appendix F of the draft standard. Where the system in the appendix is incomplete, we have referred to the logic presented in [Woodcock & Brien 92], which is similar in nature. Many of the rules are standard from set theory and first-order logic, but there are a number of additional rules, predominantly ones that deal with schemas.

4.2 Implementational Issues

Many of the inference rules in the draft standard were straightforward to implement. For instance, the rule $(\vee\vdash)$ is represented in Z-in-Isabelle as:

```
disjL          "[| $H, P, $G |- $E; $H, Q, $G |- $E |] ==>
                 $H, P \lor Q, $G |- $E"
```

The list of premises is enclosed in "`[| |]`", and premises are separated by semi-colons. The symbol "`$`" signifies that the variable it precedes may unify with a list of formulae. This Z-in-Isabelle rule corresponds to

$$\frac{\Gamma, P, \Delta \vdash \Lambda \quad \Gamma, Q, \Delta \vdash \Lambda}{\Gamma, P \vee Q, \Delta \vdash \Lambda} \; [\,(\vee\vdash)\,]$$

which is the rule $(\vee\vdash)$ in the draft Z standard (before "rule-lifting").

A number of the inference rules in the draft Z standard, however, posed problems for the implementation, either because they could not be implemented directly, or because they were awkward to use. This is discussed in the following subsections.

Bound variables and substitution. One major design decision was to exploit higher-order abstract syntax, an efficient and elegant approach to dealing with variable binding and substitution, as offered by Isabelle. Making higher-order abstract syntax applicable required some adaptations.

Many of the inference rules of Z, in particular those that involve binding constructs (universal and existential quantification, and set comprehension[3]), have side conditions to avoid variable capture. Isabelle allows no such side conditions, which means that they must be enforced directly in the way the inference rule is expressed. As explained above in section 2, higher-order abstract syntax enables us to do this by lifting the variable capture problem from the object level to the meta-level. To exploit higher-order abstract syntax, we must ensure that meta-level substitution is sound also at the object-level. This requires that any expression being substituted must be represented explicitly. This is not always the case in Z, e.g., when we use schema references (i.e., the use of the name of a schema to represent its definition). Take, for instance, the goal

$$\vdash x \in \{x : X \mid S \bullet G(x)\} \ ,$$

where the schema S is defined as:

$$\begin{array}{|l} \multicolumn{1}{l}{S} \\ \hline x : X \\ \hline Q(x) \\ \hline \end{array}$$

If we first apply the *compre* rule directly

$$\frac{\vdash \exists\, x : T \mid P(x) \bullet t = F(x)}{\vdash t \in \{x : T \mid P(x) \bullet F(x)\}} \quad [\ compre\ ]$$

we get

$$\vdash \exists\, x_0 : X \mid S \bullet x = G(x_0) \ ,$$

and if we then expand the schema by replacing its reference with its definition, we obtain

$$\vdash \exists\, x_0 : X \mid x \in X \wedge Q(x) \bullet x = G(x_0) \ ,$$

which is far from correct. The incorrectness occurred because the representation of the schema S, i.e., its name S, contains no explicit reference to the variable x. For substitution to be sound, we must therefore expand all schema references from the outset. For large specifications, this can result in long formulae and affect efficiency noticeably. The problem has been alleviated at least visually by making the name of a schema its concrete syntax. The expansion is thus only apparent in the abstract syntax.

Given that we are using higher-order abstract syntax, we would like to exploit the conveniences thereof as far as possible. In particular, we would like to use meta-level rather than object-level substitution in the rules that use explicit bindings. These are $(\forall \vdash)$, $(\vdash \exists)$ and *Leibniz*. To illustrate, here are the two versions of *Leibniz*[4]:

[3]The syntax of Z for binding constructs is not standard: $\forall\, x : S \mid P \bullet Q$ is the same as $\forall\, x : S.\ P \Rightarrow Q$, $\exists\, x : S \mid P \bullet Q$ the same as $\forall\, x : S.\ P \wedge Q$, and $\{x : S \mid P \bullet F\}$ the same as $\{x' : S \mid (\exists\, x : S.\ P \wedge x' = F)\}$.

[4]Binding application is represented as $\odot$; bindings are represented as $\langle\!|n_1 \leadsto t_1, \ldots, n_m \leadsto t_m|\!\rangle$.

$$\frac{s = t, \langle\!| x \leadsto t |\!\rangle \odot P \vdash}{s = t, \langle\!| x \leadsto s |\!\rangle \odot P \vdash}\ [\ \mathit{Leibniz}\ ] \qquad \frac{s = t, P(s) \vdash}{s = t, P(t) \vdash}\ [\ \mathit{Leibniz_1}\ ]$$

The rule *Leibniz* is the rule in the draft standard, using object-level substitution, and *Leibniz₁* is the same rule using meta-level substitution (via function application). *Leibniz* is particularly awkward to use because it often requires an application of the *cut* rule to rewrite the goal to the appropriate form $\langle\!| x \leadsto s |\!\rangle \odot P$.

The rules with explicit bindings require the substitution to be "computed" at the object-level by repeatedly applying the substitution rules. The rules with meta-level substitution lift substitution from the object to the meta-level. For this lifting to be correct, we need to establish that meta- and object-level substitution are equivalent. What we gain by this deviation from the principle of a deep embedding are significantly shorter and simpler proofs. In the following, we will use the rules using meta-level substitution (and rules derived from them) precisely to avoid overly lengthy and tedious proofs. However, both versions, i.e., with and without object-level substitutions, are available in Z-in-Isabelle.

More suitable rules of inference. As we discovered in the course of our first proof attempts, a number of the rules are difficult to use in interactive or automated proofs. We have therefore derived, in Isabelle itself, a set of rules which are easier to use, more efficient, and more suitable to automation.

Many of the inference rules in the Z standard are written as bidirectional rules, i.e., they can be applied top-to-bottom or bottom-to-top. The *compre* rule above is an example of such a rule. Such bidirectional rules are particularly awkward when writing tactics. First, there is a danger of looping. Second, whenever we wish to apply an inference to a formula on the side of the sequent symbol opposite to the one for which a rule for that type of formula is defined, we must resort to the *cut* rule. In automated theorem proving, applying the *cut* rule is generally avoided whenever possible, since it allows too great a degree of freedom. This second point is best demonstrated by an example. Let the current goal be

$$\vdash n \in \{x : \mathbb{Z} \bullet 2x\} \vdash n > 1\ .$$

To prove this goal, we need to apply the *compre* rule. However, it only applies to formulae on the right-hand side of the sequent symbol. Thus, we first need to apply the *cut* rule with the formula $\exists\, x : T \bullet n = 2x$, so that we can then apply the *compre* rule bottom-to-top. The following proof tree illustrates the process:

$$\dfrac{\dfrac{}{n \in \{x : \mathbb{Z} \bullet 2x\} \vdash \exists\, x : \mathbb{Z} \bullet n = 2x}\,\mathit{compre} \qquad \dfrac{\vdots}{\exists\, x : \mathbb{Z} \bullet n = 2x \vdash n > 1}}{n \in \{x : \mathbb{Z} \bullet 2x\} \vdash n > 1}\,\mathit{cut}$$

To avoid applying the *cut* rule, we have therefore derived decompositional left and right rules. These decompose a given formula as far as deterministically possible. Thus, for instance, from the *compre* rule (and the rules $(\vdash \exists)$, $(\exists \vdash)$, and *cut*), we have derived the following rules:

$$\frac{\vdash x : T \quad \vdash P(x) \quad \vdash t = F(x)}{\vdash t \in \{x : T \mid P(x) \bullet F(x)\}}\ [\ \mathit{compreR}\ ]$$

$$\frac{(\bigwedge x.\ x \in T, P(x), t = F(x)) \vdash}{t \in \{x : T \mid P(x) \bullet F(x)\} \vdash} \quad [\ compreL\ ]$$

The rule *compreL* can be applied directly to our goal above. We thus avoid applying the *cut* rule, and at the same time reduce the number of steps in the proof.

An alternative to derived rules are tactics. Using a tactic rather than a derived rule of inference means that the sequence of inferences specified by the tactic is executed each time it is invoked. For the derived rule, the sequence of inferences to prove it is executed only once when the rule is proven. Therefore the use of derived rules is more efficient. However, derived rules presuppose a fixed sequence of inferences. Tactics are more flexible in that they can take parameters and make choices, backtrack, etc. The key to automating proofs is thus a mix of derived rules of inference and tactics, using derived rules when there is no choice, and tactics when there is.

For the core inference rules of Z, we have derived, in Z-in-Isabelle, a large set of inference rules such as *compreR* and *compreL* above. For each definition in the toolkit, we have derived a set of lemmas which ease the proving of the all-too-frequent well-typedness goals and the unfolding of definitions. We are currently developing tactics that derive toolkit rules automatically in a systematic way. Given the size of the toolkit, this is imperative.

5 An Example Proof: Data Refinement in Z

Data refinement involves finding a concrete data type to implement the abstract data type used in a specification, defining the relationship between the two data types, and proving that the refinement is correct. In Z, the concrete data type and the relationship between the abstract and the concrete data type are usually defined in schemas. For *BirthdayBook*, [Spivey 92] selects two arrays as the concrete data type, where arrays are represented as functions from the natural numbers to the corresponding sets. *BirthdayBook1*[5] implements the concrete data type, *Abstr* shows how the abstract and concrete data types are related, and *AddBirthday1*[6] implements the concrete operation.

BirthdayBook1

$names : \mathbb{N}_1 \rightarrow NAME$
$dates : \mathbb{N}_1 \rightarrow DATE$
$hwm : \mathbb{N}$

$\forall i, j : 1 \mathrel{..} hwm \bullet i \neq j \Rightarrow names(i) \neq names(j)$

Abstr

$BirthdayBook\ ;\ BirthdayBook1$

$known = \{i : 1 \mathrel{..} hwm \bullet names(i)\}$
$\forall i : 1 \mathrel{..} hwm \bullet birthday(names(i)) = dates(i)$

[5]The identifier *hwm* stands for "high water mark".
[6]The symbol $\oplus$ represents functional overriding.

$$
\begin{array}{|l}
\hline
\textit{AddBirthday1} \\
\Delta \textit{BirthdayBook1} \\
\textit{names?} : \textit{NAME} \\
\textit{date?} : \textit{DATE} \\
\hline
\forall i : 1 \mathrel{..} \textit{hwm} \bullet \textit{name?} \neq \textit{names}(i) \\
\textit{hwm}' = \textit{hwm} + 1 \\
\textit{names}' = \textit{names} \oplus \{\textit{hwm}' \mapsto \textit{name?}\} \\
\textit{dates}' = \textit{dates} \oplus \{\textit{hwm}' \mapsto \textit{date?}\} \\
\hline
\end{array}
$$

Following [Spivey 92], we have two major proof obligations to show that a concrete operation is correct with respect to the abstract one. First, we need to establish that, given a relation (Abs) between the abstract and concrete data types ($Astate$ and $Cstate$, respectively), the pre-condition of the concrete operation (pre Cop) must be satisfied whenever the pre-condition of the abstract operation (pre Aop) is also satisfied.

$$\forall\, Astate\ ;\ Cstate\ ;\ x? : X \bullet \mathrm{pre}\ Aop \wedge Abs \Rightarrow \mathrm{pre}\ Cop.$$

Second, the state after the concrete operation ($Cstate'$) must represent one of the abstract states in which the abstract operation could terminate. Thus, given a relation Abs between the abstract and concrete data types, every possible state after the concrete operation must be related by Abs' to a possible state after the abstract operation whenever both the abstract and concrete operations are guaranteed to terminate.

$$\begin{array}{l}\forall\, Astate\ ;\ Cstate\ ;\ Cstate'\ ;\ x? : X,\ y! : Y \bullet \\ \qquad \mathrm{pre}\ Aop \wedge Abs \wedge Cop \Rightarrow (\exists\, Astate' \bullet Abs' \wedge Aop).\end{array}$$

We now show the proof of the first of the above two obligations in Z-in-Isabelle. The proof of the second is of similar difficulty, but much larger. The theorem we need to prove is

$$\begin{array}{l}\forall\, BirthdayBook\ ;\ BirthdayBook1\ ;\ names? : NAME\ ;\ date? : DATE \bullet \\ \qquad \mathrm{pre}\ AddBirthday \wedge Abs \Rightarrow \mathrm{pre}\ AddBirthday1.\end{array}$$

The initial proof state (after expanding all schema references[7]) in Z-in-Isabelle is

$$\begin{array}{ll} 1. & known : \mathbb{P}\, NAME,\ birthday : NAME \rightarrow\!\!\!\!\!\!+\ DATE, \\ & known = \mathrm{dom}\, birthday,\ names : \mathbb{N}_1 \rightarrow NAME, \\ & dates : \mathbb{N}_1 \rightarrow DATE,\ hwm : \mathbb{N}, \\ & \forall i : 1 \mathrel{..} hwm \mid true \bullet \forall j : 1 \mathrel{..} hwm \mid true \bullet \\ & \qquad i \neq j \Rightarrow names(i) \neq names(j), \\ & name? : NAME,\ date? : DATE \\ & \vdash preAddBirthday \wedge Abstr \Rightarrow preAddBirthday1 \end{array}$$

We first apply a deterministic tactic that applies all possible propositional rules and "unpacks" the schemas[8]. This produces six subgoals; all but the last are trivial and can be solved with simple deterministic tactics. The main subgoal, with the main

[7] Remember that schema names are simply the concrete syntax for the schema definition.

[8] "Unpacking" a schema means decomposing it into its constituents. A schema $[x : T \mid P(x)]$ used as a predicate decomposes into $x \in T \wedge P(x)$.

hypotheses boxed, is:

$$\begin{array}{ll}
\mathit{1.} & known : \mathbb{P}\, NAME, birthday : NAME \rightarrow\!\!\!\!\!\!+\; DATE, \\
& names : \mathbb{N}_{\mathit{1}} \rightarrow NAME, dates : \mathbb{N}_{\mathit{1}} \rightarrow DATE, hwm : \mathbb{N}, \\
& name? : NAME, date? : DATE, name? \in NAME, date? \in DATE, \\
& \boxed{name? \notin known}, known \in \mathbb{P}\, NAME, \\
& birthday \in NAME \rightarrow\!\!\!\!\!\!+\; DATE, names \in \mathbb{N}_{\mathit{1}} \rightarrow NAME, \\
& dates \in \mathbb{N}_{\mathit{1}} \rightarrow DATE, hwm \in \mathbb{N}, \\
& \forall\, i : \mathit{1} \mathrel{..} hwm \mid true \bullet \forall\, j : \mathit{1} \mathrel{..} hwm \mid true \bullet \\
& \qquad i \neq j \Rightarrow names(i) \neq names(j), \\
& known = \mathrm{dom}\, birthday, \\
& \boxed{known = \{i : \mathit{1} \mathrel{..} hwm \mid true \bullet names(i)\}}, \\
& \forall\, i : \mathit{1} \mathrel{..} hwm \mid true \bullet birthday(names(i)) = dates(i) \\
& \vdash \forall\, i : \mathit{1} \mathrel{..} hwm \mid true \bullet name? \neq names(i)
\end{array}$$

The obvious choice is to apply ($\vdash \forall$). Again, we solve all trivial subgoals. This leaves us with one subgoal (for the purpose of clarity and brevity, we omit all but the relevant hypotheses).

$$\begin{array}{ll}
\mathit{1.} & !!x. \ldots, \\
& name? \notin known, \ldots, \\
& known = \{i : \mathit{1} \mathrel{..} hwm \mid true \bullet names(i)\}, \ldots, \\
& x : \mathit{1} \mathrel{..} hwm \\
& \vdash name? \neq names(x)
\end{array}$$

To progress, we need to unfold some definitions, in particular those of $\notin$ and $\neq$[9]. While the unfolding of $\neq$ is the most obvious step for the conclusion, the fact that we need to unfold $\notin$ is not quite as straightforward to recognize. After unfolding, we apply ($\vdash \neg$) and ($\neg$ $\vdash$). This produces a number of subgoals involving set membership, which we ignore for the moment. We concentrate on our main subgoal, the sixth.

$$\begin{array}{ll}
\mathit{6.} & !!\, x. \ldots, \\
& known = \{i : \mathit{1} \mathrel{..} hwm \mid true \bullet names(i)\}, \ldots, \\
& x : \mathit{1} \mathrel{..} hwm, \\
& name? = names(x) \\
& \vdash name? \in known
\end{array}$$

The next step is crucial, yet even less obvious than the above. We need to select the right equations to substitute for *name?* and *known* in the goal. We must substitute $names(x)$ for $name?$ and $\{i : \mathit{1} \mathrel{..} hwm \mid true \bullet names(i)\}$ for *known*. We again assume that trivial subgoals are solved.

$$\begin{array}{ll}
\mathit{6.} & !!\, x. \ldots, \\
& known = \{i : \mathit{1} \mathrel{..} hwm \mid true \bullet names(i)\}, \ldots, \\
& x : \mathit{1} \mathrel{..} hwm, \\
& name? = names(x) \\
& \vdash names(x) \in \{i : \mathit{1} \mathrel{..} hwm \mid true \bullet names(i)\}
\end{array}$$

[9] $x \notin S$ is defined as $\neg\, x \in S$ and $x \neq y$ is defined as $\neg\, x = y$.

Applying the *compreR* rule (see section 4.2) yields the three subgoals

$$
\begin{array}{ll}
6. & !!\ x \ldots, \\
 & x : 1 \mathinner{\ldotp\ldotp} hwm \ldots, \\
 & \vdash ?x101(x) \in 1 \mathinner{\ldotp\ldotp} hwm \\
7. & !!\ x \ldots, \\
 & \vdash true \\
8. & !!\ x \ldots, \\
 & \vdash names(x) = names(?x101(x))
\end{array}
$$

These are trivial. Appealing to the declaration in 6 or applying reflection to 8 will instantiate the schematic expression $?x101(x)$ introduced by the *compreR* to x.

This has completed the interesting part of the proof. The remaining subgoals all involve showing well-typedness, i.e., demonstrating that something is a member of a set. While the majority of them are not difficult and can be solved with simple tactics, one is not only tedious but quite difficult, since it involves unpacking a number of definitions and some equational reasoning. The proof of this one well-typedness subgoal actually constitutes the largest (!) part of the entire proof tree.

The example proof above is but a toy example, chosen because it is short enough to present here. Nevertheless, our experience with other similar proofs in Z-in-Isabelle has shown that it is typical of larger verification proofs in Z in many respects:

1. The main branch of the proof is only a small part of the proof tree. The largest part of the tree consists of subsidiary proofs, usually to establish the well-typedness of some expression. These subsidiary proofs are often, but not always, trivial, which means that they cannot always be solved by simple brute-force tactics. While we think it is reasonable to expect users to provide some hints to the proof tool in the main branch of the proof, we cannot expect them to spend the majority of their effort doing unintuitive and tedious well-typedness proofs. Therefore, a great deal of effort must go into developing powerful tactics to deal with such subgoals fully automatically.
2. The proof states are large, to the point they no longer fit on the screen. This is mostly due to long lists of hypotheses, most of which are irrelevant to the goal at hand. We have been able to compact the list of hypotheses somewhat by applying a tactic that thins duplicate hypotheses. This is particularly useful after unpacking schemas, which often produces multiple copies of a hypothesis (*AddBirthday*, for instance, imports *BirthdayBook*, which means that the constraints of *BirthdayBook* appear in both schemas). It would also be helpful to mark the hypotheses relevant to the current subgoal, so that users can locate them easily. This is another potentially difficult problem, but a simple first approach could be to rearrange the list of hypotheses according to some measure of relevancy.
3. The set of derived inference rules that we have used has proven invaluable. In particular, we have managed to avoid resorting to the *cut* rule (though it was used often enough in proving the derived rules!), and the number of applications of other rules without the subformula property (mainly *Leibniz*) is in fact quite small. This is encouraging from the point of view of automating proofs, since avoiding such rules cuts down the search space considerably.
4. The "overhead" of using a meta-logic is quite reasonable. One concern is to what extent the use of a meta-logic is visible and distracting to users. In Z-in-Isabelle, the meta-logic is rarely apparent, and Z-in-Isabelle proofs mimic the corresponding object-level proofs. Another major concern when using a meta-logic is efficiency. While using a meta-logic cannot be expected to be as efficient

as a direct implementation, it must be efficient enough not to annoy users overly for lack of speed. Z-in-Isabelle is in fact reasonably efficient. This is not least due to the time saved by exploiting derived rules. While developing the above proof was a matter of hours, replaying the proof script is a matter of less than three minutes.

6 Conclusions

The idea underlying our research project is that formal methods will become indispensable in software engineering, and that the wide-spread acceptance and use of formal methods will crucially depend on the availability of powerful proof tools. Such a proof tool must be both reliable and carry large parts of the proof burden. While users can expect to provide assistance in the main branches of the proof, side-branches of proofs must be handled automatically.

Logical frameworks are a good foundation for such proof tools. Using a logical framework, we can break down the development of a proof tool into various steps, each of which can be checked. Starting with the syntax and semantics of a formal language, a deductive system is developed that is then proven sound and complete. Then, in a next step, the deductive system is encoded in the meta-logic of the logical framework, and the encoding is proven faithful and adequate. A deductive system which is relatively straightforward to prove sound and complete and to encode in a meta-logic is not necessarily one that is suitable for automating proofs. Given a suitable logical framework, we can, however, within the framework, develop a set of derived rules that forms a sound basis for automated proofs. We thus obtain an implementation of a formal language that is both highly reliable and suitable for automation.

However, not every logical framework and not every embedding within a logical framework forms a useful basis for a proof tool. Properties that we have found absolutely necessary are schematic variables, the possibility of proving lemmas and using them as derived rules of inference, and higher-order abstract syntax. While the latter has required us to sacrifice some deepness of embedding for the sake of correctness, we believe that the benefits, i.e., efficiency and simplicity, are worth the cost. Another deep embedding of Z in a logical framework is Jigsaw, which is implemented in 2OBJ [Goguen *et al* 92]. While Jigsaw is the deeper embedding, it is too slow to be useful. There are two main reasons for this. First, Jigsaw must rely on tactics where Z-in-Isabelle uses derived rules, since 2OBJ does not support derived rules. Second, Jigsaw does not use higher-order abstract syntax, wherefore a large amount of computation is spent in explicitly computing side conditions and substitutions.

The results we have achieved so far in Z-in-Isabelle are promising. Z-in-Isabelle is a sound basis on which a proof tool for reasoning about Z specifications can be built. While the system now requires an experienced user of theorem proving systems to do proofs, we believe that a system that can be used by software engineers is ultimately within reach. The main steps in that direction are providing support for users so that they do not get lost in proof states or even proof trees, and automating subsidiary proofs. Tackling these problems is therefore our foremost priority for future work.

References

[Bowen & Gordon 94] J.P. Bowen and M.J.C. Gordon. Z and HOL. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 141–167. Springer-Verlag, 1994.

[Brien & Nicholls 92] S. M. Brien and J. E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, November 1992. Accepted for ISO standardization, ISO/IEC JTC1/SC22.

[Goguen *et al* 92] J. Goguen, A. Stevens, H. Hilberdink, and K. Hobley. 2OBJ: A metalogical theorem prover based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992.

[Harwood 91] W. T. Harwood. Proof rules for Balzac. Technical Report WTH/P7/001, Imperial Software Technology, Cambridge, UK, 1991.

[Jones 92] R. B. Jones. ICL ProofPower. *BCS FACS FACTS*, Series III, 1(1):10–13, Winter 1992.

[Jordan 91] L. E. Jordan. The Z syntax supported by Balzac-II/1. Technical Report LEJ/S1/001, Imperial Software Technology, Cambridge, UK, 1991.

[Maharaj 90] S. Maharaj. Implementing Z in Lego. Unpublished M.Sc. thesis, Department of Computer Science, University of Edinburgh, September 1990.

[Martin 94] A. Martin. *Machine-Assisted Theorem-Proving for Software Engineering.* Unpublished PhD thesis, University of Oxford, 1994.

[Paulson 89] L.C. Paulson. The foundation of a generic theorem prover. *The Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Paulson 94] L.C. Paulson. *Isabelle. A Generic Theorem Prover.* Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Pfenning & Elliot 88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[Spivey 92] J. M. Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, 2nd edition, 1992.

[Woodcock & Brien 92] J. C. P. Woodcock and S. M. Brien. W: A logic for Z. In J. E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 77–96. Springer-Verlag, 1992.