Z and HOL

Jonathan Bowen
Oxford University
Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD
UK
Email: Jonathan.Bowen@comlab.ox.ac.uk

Mike Gordon
University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
UK
Email: Mike.Gordon@cl.cam.ac.uk

Abstract

A simple 'shallow' semantic embedding of the Z notation into the HOL logic is described. The Z notation is based on set theory and first order predicate logic. The HOL theorem proving system supports higher order logic. A well-known case study is used as a running example. The presentation is intended to show people with some knowledge of Z how a tool such as HOL can be used to provide mechanical support for the notation, including mechanization of proofs. No specialized knowledge of HOL is assumed.

1 Introduction

HOL [8] is an 'LCF-style' theorem proving environment [9, 21] for classical higher order logic [1, 5]. Proof tools for the formal specification notation Z [4, 29] can be implemented by translating Z schemas into higher order logic and then programming schema combining operations in HOL's metalanguage ML.[1] This technique is known as *semantic embedding* (see section 3.2) and has been pioneered by ICL for many years. They have built their own 'industrial strength' version of HOL, called ProofPower [12], and implemented sophisticated proof tools for Z on top of it (see section 4 for more on ProofPower and other proof tools for Z).

This paper is intended to be an accessible introduction to support for Z by semantic embedding. It is intended for readers familiar with Z, but not with HOL. The techniques described here use ordinary HOL and are simpler and less powerful than those found in ProofPower. In the next section a familiar Z specification example, Spivey's 'birthday book', is used to illustrate how HOL can support Z. Subsequent sections discuss how this support is achieved.

2 The Birthday Book

It is assumed that readers are reasonably familiar with the birthday book example in Chapter 1 of Spivey's *The Z Notation: A Reference Manual* [29] (also to

[1] The ML language was originally developed as part of LCF, but is now an independent programming language in its own right [18]. It is an eager-evaluation impure functional language with a polymorphic type discipline.

be found in [27]), henceforth called ZRM. The schemas from this specification are included here for comparison with their HOL equivalents.

The numbered boxes that follow show a HOL session in which the birthday book is input and some simple facts are proved. The HOL system prompts for input with `#`, so all lines beginning with this symbol are supplied by the user. All other lines are generated by HOL. The sessions have been edited to remove some output and to suppress details of some user-supplied theorem proving tactics, which will only be comprehensible to readers familiar with HOL.[2] Before the session starts HOL is run and the Z support package loaded. This package is a collection of ML declarations together with a theory `Z` containing definitions to support Z operators and its 'Mathematical Tool-kit' (see section 3.3). Details of this loading are omitted here.

The first interaction of the session is shown in box 1 below. A new theory called `BirthdayBook` is started. Theories contain definitions and theorems that the user has proved. They are stored in a hierarchical database on disk. The ML command `new_theory` starts a new theory with a user-supplied name.

```
#new_theory `BirthdayBook`;;                                          1
```

This interaction initiates the specification of the Birthday Book by starting a new HOL theory called `BirthdayBook`. Note that in the version of ML (Classic ML) provided by HOL88 (which is the version of HOL used here) user input is terminated with `;;` (in Standard ML the terminator is a single semi-colon).

The first definition in this theory introduces two sets:

$$[\ NAME, DATE\ ]$$

The corresponding declaration in HOL uses an ML function `sets`.

```
#sets `NAME DATE`;;                                                   2
```

With this second interaction the two sets `NAME` and `DATE` are declared.

The first schema in the Birthday Book defines the abstract state and an 'invariant' on that state:

BirthdayBook

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

$known = \operatorname{dom} birthday$

The declaration of schemas in HOL uses a notation that is intended to make it clear what the corresponding Z is. This is shown in box 3 below.

[2] The complete input to HOL for the Birthday Book and other examples is available in the directory `contrib/Examples` distributed with HOL88, Version 2.02. Details of HOL (including how to get it) are available from an on-line networked hypertext documentation service run by the Laboratory for Applied Logic at Brigham Young University. To browse this documentation, first obtain the XMosaic tool (connect to `ftp.ncsa.uiuc.edu` by anonymous FTP, then the directory `/Mosaic/xmosaic-binaries` contains compiled copies of the latest version of XMosaic for a variety of architectures). To connect to the HOL documentation server select `OPEN` from the XMosaic menu and enter `http://lal.cs.byu.edu/lal/hol-documentation.html`.

```
#declare                                                    3
# `BirthdayBook`
# "SCHEMA
#   [known :: (P NAME);
#    birthday :: (NAME -+> DATE)]
#   %--------------------------%
#   [known = dom birthday]";;
```

This third interaction illustrates the format (shown on lines starting with `#`) used to input Z schemas to HOL. A preprocessor converts the input format to a HOL term that represents the semantics of the corresponding schema. This preprocessor could in principle be a complete parser for Z notation,[3] but in keeping with our 'lightweight' approach this has not been done (ICL's ProofPower tool supports proper Z syntax). The aim has been to have an input format that is readable as both Z and HOL.

The powerset operator `P` and infix partial function operator `-+>` are defined in the theory `Z`, which implements (part of) the Z mathematical tool-kit; see section 3.3 for more details.

The ML function `declare` converts the term `"SCHEMA` $\cdots$`"` that follows it into the HOL representation of the schema, remembers the declaration (and the types of declared variables) in a global data structure and returns the schema representation.

The HOL representation of a schema of the form:

$$
\begin{array}{|l}
\cdots \\
x_1 : S_1 \\
x_2 : S_2 \\
\quad\vdots \\
x_m : S_m \\
\hline
P_1 \\
P_2 \\
\vdots \\
P_n
\end{array}
$$

is semantically equivalent to the formula:

$$x_1 \in S_1 \wedge x_2 \in S_2 \wedge \ldots \wedge x_m \in S_m \wedge P_1 \wedge P_2 \wedge \ldots \wedge P_n$$

which in HOL notation is written as:

x_1 `IN` S_1 `/\` x_2 `IN` S_2 `/\ ... /\` x_m `IN` S_m `/\` P_1 `/\` P_2 `/\ ... /\` P_n

where S_i is a HOL term denoting a set and $P_1, \ldots, P_n$ are Boolean terms constituting the schema's predicate. The infix operators `IN` and `::` both denote the set membership relation. The former is used for general set membership in predicates (e.g. to represent $\in$), the latter for type membership assertions (e.g. in the declaration part of schemas, where Z would have a colon). Note that

[3] HOL libraries provide syntax processing utilities that enable custom parsers and pretty-printers to be generated.

schema operations can move type membership assertions from the declaration part of schemas into the predicate part, thus automatically generated assertions of the form $x :: S$ may appear in the predicate parts of schemas.

The actual semantic representation is not the conjunction shown above, but the logically equivalent term:

```
SCHEMA [x1 :: S1; x2 :: S2; ... ;xm :: Sm]
       [P1; P2; ... ; Pn]
```

where `SCHEMA` is defined (in the theory `Z`) by:

```
SCHEMA [d1; ... ;dm] [P1; ... ;Pn]  =
 d1 /\ ...  /\ dm /\ P1 /\ ...  /\ Pn
```

This representation is more convenient as the conjuncts corresponding to the declaration and predicate parts of the schema can be easily extracted.

HOL input will usually be laid out in the following 'Z-style' format:

```
SCHEMA
 [x1  :: S1;
  x2  :: S2;
        ⋮
  xm  :: Sm]
 %---------%
 [P1;
  P2;
   ⋮
  Pn]
```

where the line `%--···--%` is just a comment to aid the eye (comments in ML are enclosed between percent symbols).

The next schema illustrates schema inclusion and the Δ change of state schema convention. First the Z:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddBirthday}} \\
\Delta \textit{BirthdayBook} \\
\textit{name?} : \textit{NAME} \\
\textit{date?} : \textit{DATE} \\
\hline
\textit{name?} \notin \textit{known} \\
\textit{birthday}' = \textit{birthday} \cup \{\textit{name?} \mapsto \textit{date?}\}
\end{array}$$

The corresponding HOL is:

```
#declare                                                    4
# `AddBirthday`
# "SCHEMA
#   [DELTA BirthdayBook;
#    name? :: NAME;
#    date? :: DATE]
#   %------------------------------------------%
#   [~(name? IN known);
#    birthday' = birthday UNION {name? |-> date?}]";;
```

The symbol ~ represents negation ($\neg$) in HOL. Normally schemas are printed out as their name, but this default can be changed so that the semantic representations of schemas are printed in full. This mode is entered by evaluating the ML expression `show_schemas true`.

```
#"BirthdayBook";;                                                        5
"BirthdayBook" : term

#show_schemas true;;

#"BirthdayBook";;
"SCHEMA
 [known :: (P NAME);
  birthday :: (NAME -+> DATE)]
 [known = dom birthday]" : term

#"AddBirthday";;
"SCHEMA
 [known :: (P NAME);
  birthday :: (NAME -+> DATE);
  known' :: (P NAME);
  birthday' :: (NAME -+> DATE);
  name? :: NAME;date? :: DATE]
 [known = dom birthday;
  known' = dom birthday';
  ~name? IN known;
  birthday' = birthday UNION {name? |-> date?}]"
```

Note how the schema inclusion `DELTA BirthdayBook` (which represents Z's Δ*BirthdayBook*) results in extra primed variables being added to the declaration of `AddBirthday`, and appropriate instances of the predicate part of `BirthdayBook` being included in the predicate of `AddBirthday`. More details of how `DELTA`-expansion is implemented can be found in section 3.1.

As a check, the precondition of `AddBirthday` can be computed and simplified using a theorem proving tool called `simp`. The precondition of an operation schema is the condition on the state that must hold if the operation is to be applicable. See page 77 of [29], page 151 of [23] and page 141 of [7] for further discussions of this. For example, here is the computation of the precondition of *AddBirthday*:

```
#show_schemas false;;                                                    6

#simp "pre AddBirthday";;
known :: (P NAME),
birthday :: (NAME -+> DATE),
name? :: NAME,
date? :: DATE,
(dom(birthday UNION {name? |-> date?})) :: (P NAME),
(birthday UNION {name? |-> date?}) :: (NAME -+> DATE)
|- pre AddBirthday = (known = dom birthday) /\ ~name? IN (dom birthday)
```

The evaluation of `simp "pre AddBirthday"` results in a theorem asserting that the precondition of *AddBirthday* is $known = \mathrm{dom}\, birthday \wedge name? \notin \mathrm{dom}\, birthday$.

Theorems are a special datatype in ML, values of which can only be created by applying sequences of inference rules to axioms (or previously proved theorems) [9, 8]. A theorem $t_1, \ldots, t_n$ `|-` t asserts that the term t follows from the conjunction of the terms $t_1, \ldots, t_n$.

In the example above, the terms before the turnstile `|-` are type inclusion assumptions used by `simp` in simplifying `pre AddBirthday`. The function `simp` uses some simple heuristics (e.g. the 'One Point Rule' [23], which is implemented in the `unwind` library), to simplify the supplied term. The heuristics are adequate here, but not for more complex examples (see `RAddBirthday` below). Special purpose theorem proving tools like `simp` are easy for users to implement for themselves in ML.

Sets are regarded as primitive in Z, but in HOL they must be defined. Fortunately, HOL's `sets` library defines a type of sets and the usual operations of set theory. It also defines the set comprehension notation:

$$\{\ E[x_1, \cdots, x_n]\ \mid P[x_1, \cdots, x_n]\ \}$$

which is equivalent to:

$$\{\ x\ \mid \exists\, x_1 \cdots x_n.\ x = E[x_1, \cdots, x_n]\ \wedge\ P[x_1, \cdots, x_n]\ \}$$

Thus the Z set comprehension notation $\{S \bullet E\}$ is translated into $\{E \mid S\}$ in HOL. It would be possible to support Z's comprehension notation directly, but as the translation is so direct this has not been done.

The theory `Z` provides the definitions of some of the set theoretic operators in Z and its mathematical toolkit. For example, Z functions are a special case of relations:

$$\begin{aligned} X \leftrightarrow Y &= \mathbb{P}(X \times Y) \\ X \mathrel{+\!\!\!\rightarrow} Y &= \{f : X \leftrightarrow Y \mid (\forall\, x : X;\ y_1, y_2 : Y \bullet \\ &\qquad (x \mapsto y_1) \in f \wedge (x \mapsto y_2) \in f \Rightarrow y_1 = y_2)\} \end{aligned}$$

The corresponding HOL definitions are:

```
X <-> Y  =  P(X >< Y)
X -+> Y  =  {f | f IN (X <-> Y) /\ (!x y1 y2.
               (x|->y1) IN f /\ (x|->y2) IN f ==> (y1 = y2))}
```

where `!` is HOL's notation for the universal quantifier $\forall$. These definitions are 'generic': `X` and `Y` range over sets of arbitrary elements. In the HOL logic functions are a primitive concept and unlike in Z are not regarded as certain kinds of sets. The HOL notation $f\,(x)$, which can also be written without brackets as just $f\ x$, denotes the application of the logical function f to argument x. If f is a set of ordered pairs, then this is not a well-formed HOL term – only logical functions can be applied to arguments. To get around this problem occurrences of $f\,(x)$, where f has been previously declared to be a function graph, are pre-processed to f`^^`x, where `^^` is an infix operator for 'applying' sets defined in the theory `Z` (see section 3.3).

The specification presented so far can be further tested by proving the property discussed on page 5 of ZRM. In Z notation this is:

$$\mathit{AddBirthday} \vdash \mathit{known}' = \mathit{known} \cup \{\mathit{name}?\}$$

In HOL the proof proceeds as follows:

```
#prove_theorem                                                          7
# (`known_UNION`,
#  "[AddBirthday] |-? (known' = known UNION name?)",
#  REWRITE_ALL_TAC[SCHEMA;CONJL;dom_UNION;dom_SING]);;

known_UNION = AddBirthday |- known' = known UNION name?
```

The input has the form `prove_theorem(`*name*`,`*goal*`,`*tactic*`)`. This instructs HOL to try to prove *goal* using *tactic*. A goal of the form `[`t_1`;`$\ldots$`;`t_n`] |-?` t is solved if *tactic* proves the theorem $t_1, \ldots, t_n$ `|-` t. A goal just consisting of a term t has no assumptions and is solved if *tactic* proves the theorem `|-` t (see the example in box 18 below).

The tactic used above rewrites the goal and assumptions with the definitions of `SCHEMA` and `CONJL` (see section 3.3) and also with the following two laws (which are pre-proved in the theory `Z`, see section 3.3).

```
dom_UNION   |- dom(X UNION Y) = (dom X) UNION (dom Y)
dom_SING    |- dom{x |-> y}   = {x}
```

If the proof succeeds, as it does here, then the resulting theorem value is bound to *name* in the metalanguage ML and may be used subsequently.

```
#known_UNION;;                                                          8
AddBirthday |- known' = known UNION {name?}

#show_schemas true;;

#known_UNION;;
SCHEMA
[known :: (P NAME);birthday :: (NAME -+> DATE);known' :: (P NAME);
 birthday' :: (NAME -+> DATE);name? :: NAME;date? :: DATE]
[known = dom birthday;
 known' = dom birthday';
 ~name? IN known;
 birthday' = birthday UNION {name? |-> date?}]
|- known' = known UNION {name?}
```

The next operation schema in the birthday book specification is:

FindBirthday
$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

This uses the Ξ (no change of state) convention in the declaration.

```
#declare                                                          9
# `FindBirthday`
# "SCHEMA
#   [XI BirthdayBook;
#    name? :: NAME;
#    date! :: DATE]
#   %----------------------%
#   [name? :: known;
#    date! = birthday(name?)]";;
```

The schema `FindBirthday` is expanded out to a term logically equivalent to:

```
SCHEMA
 [known :: (P NAME);
  birthday :: (NAME -+> DATE);
  known' :: (P NAME);
  birthday' :: (NAME -+> DATE);
  name? :: NAME;
  date! :: DATE]
 [known = dom birthday;
  known' = known;
  birthday' = birthday;
  name? :: known;
  date! = birthday ^^ name?]" : term
```

The actual expansion is described in section 3.1.

A second lemma to check the specification can now be proved. In Z notation it is:

$$(AddBirthday \fatsemi FindBirthday) \vdash (date! = date?)$$

This checks that a subsequent *FindBirthday* operation results in the same date for a given name as that provided by a previous *AddBirthday* operation.

The hypothesis is the sequential composition of two schemas. This illustrates the shallowness of the embedding of Z. The composition of schemas is computed when they are input by the preprocessor: `SEQ` acts like a macro. With a *deep embedding* (see section 3.2) the composition operator `SEQ` would have to be defined in the object logic.

```
#prove_theorem                                                    10
# (`SEQ_AddBirthday_FindBirthday`,
#  "[AddBirthday SEQ FindBirthday] |-? (date! = date?)",
#  REWRITE_ALL_TAC[SCHEMA;CONJL]
#   THEN POP_ASSUM STRIP_ASSUME_TAC
#   THEN SMART_ELIMINATE_TAC
#   THEN IMP_RES_TAC Ap_UNION2
#   THEN ASM_REWRITE_TAC[]);;

SEQ_AddBirthday_FindBirthday =
SCHEMA
[known :: (P NAME);birthday :: (NAME -+> DATE);known' :: (P NAME);
 birthday' :: (NAME -+> DATE);name? :: NAME;
 date? :: DATE;date! :: DATE]
[?known'' birthday''.
  (known'' :: (P NAME) /\ birthday'' :: (NAME -+> DATE)) /\
  (known = dom birthday) /\
  (known'' = dom birthday'') /\
  ~name? IN known /\
  (birthday'' = birthday UNION name? |-> date?) /\
  (known'' = dom birthday'') /\
  (known' = dom birthday') /\
  ((`birthday`,birthday'),`known`,known' =
   (`birthday`,birthday''),`known`,known'') /\
  name? :: known'' /\
  (date! = birthday'' ^^ name?)]
|- date! = date?
```

The symbol `?` is HOL's notation for the universal quantifier $\exists$. Iterated quantifications of the form $Q\ x_1 \cdots x_n\,.\,t$ and $Q\ x_1 \cdots x_n :: S\,.\,t$ are allowed (where Q is either `!` or `?`).

The tactic above has the form tac_1 `THEN` tac_2 `THEN` tac_3 `THEN` tac_5 `THEN` tac_5, which instructs HOL to apply $tac_1 \ldots tac_5$ in that order:

- tac_1 rewrites the goal with the definitions of `SCHEMA` and `CONJL`;
- tac_2 simplifies and 'explodes' the assumption;
- tac_3 is `SMART_ELIMINATE_TAC` which removes redundant assumptions (it was contributed [4] by Donald Syme of the Australian National University);
- tac_4 tries to combine the preproved law `Ap_UNION2` with assumptions using Modus Ponens, where `Ap_UNION2` is:

```
|-  ~x IN (dom X) ==> ((X UNION {x |-> v}) ^^ x = v)
```

- tac_5 rewrites with all the current assumptions (including any extra ones generated by tac_4).

Switching off schema printing causes the input form to be reconstructed. (How this is done is explained in section 3.1.)

[4] HOL is distributed with both a library and a `contrib` directory. The former contains uniformly documented material; the latter contains less formal contributions including `SMART_ELIMINATE_TAC`.

```
#show_schemas false;;                                              11

#SEQ_AddBirthday_FindBirthday;;
AddBirthday SEQ FindBirthday |- date! = date?
```

The next two schemas given in ZRM (*Remind* and *InitBirthdayBook*) do not add anything new and are omitted.

The birthday book uses a 'free set' called *REPORT*, defined in Z by:

$$REPORT ::= ok \mid already\_known \mid not\_known$$

Such definitions are supported in HOL using a type definition facility written by Tom Melham [17] (which is slightly repackaged for use with Z).

```
#free_set `REPORT = ok | already_known | not_known`;;              12
```

Melham's package only supports a subset of Z's free types. A wider class is definable in Isabelle's ZF application using a fixpoint method developed by Paulson [22]. This approach suggests a way of handling more of Z's free types in HOL; see also [26].

Next a couple of routine schema declarations to handle success and error conditions:

Success
---
result! : *REPORT*
---
result! = *ok*

AlreadyKnown

Ξ*BirthdayBook*
name? : *NAME*
result! : *REPORT*
---
name? ∈ *known*

result! = *already_known*

These are input to HOL by:

```
#declare                                                           13
# `Success`
# "SCHEMA
#   [result! :: REPORT]
#   %-----------------%
#   [result! = ok]";;
```

and

```
#declare                                                           14
# `AlreadyKnown`
# "SCHEMA
#   [XI BirthdayBook;
#    name? :: NAME;
#    result! :: REPORT]
#   %-----------------------%
#   [name? :: known;
#    result! = already_known]";;
```

The next schema in the birthday book uses schema conjunction and disjunction:

$$RAddBirthday \mathrel{\widehat=} (AddBirthday \wedge Success) \vee AlreadyKnown$$

The conjunction and disjunction of schemas is computed on input, just as for the sequential composition `SEQ`. `AND` and `OR` can be regarded as macros.

15

```
#declare
# `RAddBirthday`
# "(AddBirthday AND Success) OR AlreadyKnown";;

"RAddBirthday" : term

#show_schemas true;;

#"RAddBirthday";;
"SCHEMA
 [known :: (P NAME);
  birthday :: (NAME -+> DATE);
  known' :: (P NAME);
  birthday' :: (NAME -+> DATE);
  name? :: NAME;
  date? :: DATE;
  result! :: REPORT]
 [(known = dom birthday) /\
  (known' = dom birthday') /\
  ~name? IN known /\
  (birthday' = birthday UNION {name? |-> date?}) /\
  (result! = ok) \/
  (known = dom birthday) /\
  (known' = known) /\
  (birthday' = birthday) /\
  name? :: known /\
  (result! = already_known)]" : term
```

Note that in HOL, `/\` binds tighter than `\/`, so the body of this schema is a disjunction of conjunctions. It is easy to prove it equivalent to:

RAddBirthday

$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$
$result! : REPORT$

$$\begin{aligned}&(name? \notin known \wedge\\ &\quad birthday' = birthday \cup \{name? \mapsto date?\} \wedge\\ &\quad result! = ok) \vee\\ &(name? \in known \wedge\\ &\quad birthday' = birthday \wedge\\ &\quad result! = already\_known)\end{aligned}$$

From now on the tactics used to prove theorems will be omitted.

16

```
#show_schemas false;;

#prove_theorem
# (`RAddBirthdayLemma`,
#  "(AddBirthday AND Success) OR AlreadyKnown =
#   SCHEMA
#    [DELTA BirthdayBook;
#     name? :: NAME;
#     date? :: DATE;
#     result! :: REPORT]
#    %-------------------------------------------------------%
#    [(~(name? IN known) /\
#              (birthday' = birthday UNION name? |-> date?) /\
#              (result! = ok)) \/
#     (name? IN known /\
#              (birthday' = birthday) /\
#              (result! = already_known))]",
#  < omitted tactic > );;

RAddBirthdayLemma =
|- RAddBirthday =
   SCHEMA
   [known :: (P NAME);birthday :: (NAME -+> DATE);known' :: (P NAME);
    birthday' :: (NAME -+> DATE);name? :: NAME;date? :: DATE;
    result! :: REPORT]
   [known = dom birthday;known' = dom birthday';
    ~name? IN known /\
    (birthday' = birthday UNION name? |-> date?) /\
    (result! = ok)
    name? IN known /\
    (birthday' = birthday) /\
    (result! = already_known)]
```

This proof confirms informal remarks made on page 9 of ZRM (page 10 in the first edition). Note that the left hand side of the equation proved is printed as `RAddBirthday` rather than as `(AddBirthday AND Success) OR AlreadyKnown`. This is because the former is recognized as being defined equal to the latter. The right hand side of the equation has not previously been given a name, so it is printed out in full.

A further check on *RAddBirthday* is to show that its precondition is true. Unfortunately, `simp` is not powerful enough:

17

```
#simp "pre RAddBirthday";;
known :: (P NAME),
birthday :: (NAME -+> DATE),
name? :: NAME,
date? :: DATE
|- pre RAddBirthday =
   (?known' birthday' result!.
     (known' :: (P NAME) /\
      birthday' :: (NAME -+> DATE) /\
      result! :: REPORT) /\
     ((known = dom birthday) /\
      (known' = dom birthday') /\
      ~name? IN known /\
      (birthday' = birthday UNION {name? |-> date?}) /\
      (result! = ok) \/
      (known = dom birthday) /\
      (known' = dom birthday') /\
      ((birthday' = birthday) /\ (known' = known)) /\
      name? :: known /\
      (result! = already_known)))
```

The heuristics employed by `simp` cannot deal with the existentially quantified disjunction. They could be improved to work for this example, but sooner or later another example would crop up that is not handled. What is required is user guided simplification. This can be achieved in various ways, the approach illustrated here is to prove that the precondition is true with a user-supplied tactic.

18

```
#prove_theorem
# (`pre_RAddBirthday`,
#  "[BirthdayBook; sig RAddBirthday] |-? pre RAddBirthday",
#  < omitted tactic > );;

pre_RAddBirthday =
BirthdayBook, sig RAddBirthday |- pre RAddBirthday
```

The `sig` of a schema consists of the type membership statements of its variables.

$$\texttt{sig}\ (\texttt{SCHEMA}\ [x_1 :: S_1;\ \ldots;\ x_n :: S_n]\ [\cdots])\ \ =\ \ x_1 \in S_1 \wedge \ldots \wedge x_n \in S_n$$

The last three schema definitions in the abstract specification of the birthday book are *NotKnown*, *RFindBirthday* and *RRemind*. These are straightforward and are omitted here.

In section 1.5 of ZRM the operations and data structures of the birthday book are implemented. The concrete operation corresponding to *BirthdayBook* is *BirthdayBook*1, where:

*BirthdayBook*1

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

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

The HOL version of this uses the terms `NN` and `NN_1` which denote the sets of natural numbers and strictly positive natural numbers, respectively. The notation $m \mathinner{.\,.} n$ denotes the set of numbers in the closed interval $[m, n]$ ("`..`" is a `HOL` infix operator). `HOL` allows restricted quantifications of the form `!`x`::`S`.`t and `?`x`::`S`.`t (where S is a term denoting a set), which are equivalent to `!`x`.`x `IN` S `==>` t and `?`x`.`x `IN` S `/\` t, respectively. The infix operator `-->` is defined in the theory `Z` and constructs the set of total functions.

Using these notations, the Z schema above is input into HOL as:

```
#declare                                                          19
# `BirthdayBook1`
# "SCHEMA
#    [names :: (NN_1-->NAME);
#     dates :: (NN_1-->DATE);
#     hwm :: NN]
#    %-----------------------------------------------------%
#    [!i j::(1..hwm). ~(i = j) ==> ~(names(i) = names(j))]";;
```

The relation between *BirthdayBook* and its implementation *BirthdayBook*1 is specified with the schema *Abs*:

Abs

BirthdayBook
*BirthdayBook*1

$known = \{\ i : 1 \mathinner{.\,.} hwm \bullet names(i)\ \}$

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

The HOL version is:

```
#declare                                                          20
# `Abs`
# "SCHEMA
#    [BirthdayBook;
#     BirthdayBook1]
#    %------------------------------------------%
#    [known = {names(i) | i::(1..hwm)};
#     !i::(1..hwm). birthday(names(i)) = dates(i)]";;
```

The implementation of *AddBirthday* is *AddBirthday*1, where:

*AddBirthday*1

$\Delta BirthdayBook1$
$name? : NAME$
$date? : DATE$

$\forall\, i : 1 \mathinner{.\,.} hwm \bullet name? \neq names(i)$

$hwm' = hwm + 1$
$names' = names \oplus \{hwm' \mapsto name?\}$
$dates' = dates \oplus \{hwm' \mapsto date?\}$

The HOL version of this is:

```
#declare                                                              21
# `AddBirthday1`
# "SCHEMA
#    [DELTA BirthdayBook1;
#     name? :: NAME;
#     date? :: DATE]
#    %----------------------------------%
#    [!i::(1..hwm). ~(name? = names(i));
#     hwm' = hwm + 1;
#     names' = names (+) {hwm' |-> name?};
#     dates' = dates (+) {hwm' |-> date?}]";;
```

The final three schemas in the implementation of the birthday book are *FindBirthday*1, *AbsCards*, *Remind*1 and *InitBirthdayBook*1. They introduce nothing new and are omitted.

The specification of the implementation of the birthday book is now complete. The soundness of the implementation can be verified by proving that the rules for operation and data refinement are met. These rules are described in sections 5.5 and 5.6 of ZRM. When applied to the birthday book, they generate the following two conditions:

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

$$\begin{array}{l} \forall\, BirthdayBook;\ BirthdayBook1;\ BirthdayBook1';\\ \quad name? : NAME;\ date? : DATE \bullet \\ \qquad \mathrm{pre}\ AddBirthday \wedge Abs \wedge AddBirthday1 \\ \qquad \Rightarrow (\exists\, BirthdayBook' \bullet Abs' \wedge AddBirthday) \end{array}$$

It is straightforward to verify these properties in HOL. The proofs are quite a bit more complex than previous ones, involving various mathematical laws (see section 3.3) and extensive case analysis. The details will not be given here, but are available in the directory `contrib/Z/examples` distributed with HOL88 Version 2.02.

The first theorem establishes that the precondition of *AddBirthday*1 is liberal enough.

```
#prove_theorem                                                        22
# (`AbsThm1`,
#  "FORALL [BirthdayBook; BirthdayBook1; (name?::NAME); (date?::DATE)]
#     ((pre AddBirthday /\ Abs) ==> (pre AddBirthday1))",
#  < omitted tactic > );;

AbsThm1 =
|- FORALL
    [BirthdayBook;BirthdayBook1;name? :: NAME;date? :: DATE]
    (pre AddBirthday /\ Abs ==> pre AddBirthday1)
```

The schema operations `pre` and `FORALL` are expanded on input. Note that `FORALL` is interpreted as yielding a predicate not a schema (the quantifier `SCHEMA_FORALL` returns a schema).

There is no standard way of formulating theorems about Z schemas (though a start is made in Annexe F of the *Z Base Standard* [4]). For example, the two theorems in Z notation above admit a variety of formal interpretations.

'pre *AddBirthday* ∧ *Abs*' could be interpreted as either a schema expression or as a predicate; in the former case ∧ is a schema operation (represented by `AND` in HOL), in the latter case it is a logical operation (`/\` in HOL). In this particular case the two interpretations of conjunction result in logically equivalent terms in HOL, but in general it is unclear if this will always be the case (particularly with implications and quantifications). The convention for theorems adopted here is, wherever possible, to interpret Z's logical operators as operators that construct predicates rather than schemas. Other choices are possible, for example the following version of `AbsThm1` is also provable:

```
FORALL
 [BirthdayBook; BirthdayBook1; (name?::NAME); (date?::DATE)]
 ((pre AddBirthday AND Abs) ==> (pre AddBirthday1))
```

The second theorem establishes that *AddBirthday*1 produces the right answer.

```
#prove_theorem                                                    23
# (`AbsThm2`,
#  "FORALL [BirthdayBook; BirthdayBook1; BirthdayBook1';
#          (name?::NAME); (date?::DATE)]
#    ((pre AddBirthday /\ Abs /\ AddBirthday1)
#     ==>
#     (EXISTS BirthdayBook' (Abs' /\ AddBirthday)))",
#  < omitted tactic > );;

AbsThm2 =
|- FORALL
   [BirthdayBook; BirthdayBook1; BirthdayBook1';
    name? :: NAME; date? :: DATE]
   (pre AddBirthday /\ Abs /\ AddBirthday1 ==>
    EXISTS BirthdayBook'(Abs' /\ AddBirthday))
```

The schema operations `pre`, `FORALL` and `EXISTS` are expanded on input. `FORALL` and `EXISTS` are interpreted as yielding predicates not schemas (the quantifier `SCHEMA_EXISTS` returns a schema). The priming conventions for schemas are also handled on input. For example,

```
#show_schemas true;;                                              24

#"BirthdayBook";;
"SCHEMA
 [known :: (P NAME);birthday :: (NAME -+> DATE)]
 [known = dom birthday]" : term

#"BirthdayBook'";;
"SCHEMA
 [known' :: (P NAME);birthday' :: (NAME -+> DATE)]
 [known' = dom birthday']" : term
```

It is a tribute to Z's notational power using the schema notation that complex theorems like *AbsThm*1 and *AbsThm*2 can be expressed concisely. This complexity is revealed if the Z printing is switched off so that the semantic representations of schemas is output as illustrated in box 25. This graphically illustrates why the schema notation was invented to structure and hide the mass of detailed mathematics which can be held in a Z specification.

```
#AbsThm2;;                                                          25
|- !known birthday.
    known :: (P NAME) /\ birthday :: (NAME -+> DATE) ==>
    known :: (P NAME) /\ birthday :: (NAME -+> DATE) /\
    (known = dom birthday) ==>
    (!names dates hwm.
      names :: (NN_1 --> NAME) /\ dates :: (NN_1 --> DATE) /\ hwm :: NN ==>
      names :: (NN_1 --> NAME) /\ dates :: (NN_1 --> DATE) /\ hwm :: NN /\
      (!i j :: 1 .. hwm. ~(i = j) ==> ~(names ^^ i = names ^^ j)) ==>
      (!names' dates' hwm'.
        names' :: (NN_1 --> NAME) /\ dates' :: (NN_1 --> DATE) /\
        hwm' :: NN ==>
        names' :: (NN_1 --> NAME) /\ dates' :: (NN_1 --> DATE) /\
        hwm' :: NN /\
        (!i j :: 1 .. hwm'. ~(i = j) ==> ~(names' ^^ i = names' ^^ j)) ==>
        (!name?.
          name? :: NAME ==>
          (!date?.
            date? :: DATE ==>
            SCHEMA
            [known :: (P NAME);birthday :: (NAME -+> DATE);
             name? :: NAME;date? :: DATE]
            [?known' birthday'.
              (known' :: (P NAME) /\ birthday' :: (NAME -+> DATE)) /\
              (known = dom birthday) /\ (known' = dom birthday') /\
              ~name? IN known /\
              (birthday' = birthday UNION {name? |-> date?})] /\
            SCHEMA
            [known :: (P NAME);birthday :: (NAME -+> DATE);
             names :: (NN_1 --> NAME);dates :: (NN_1 --> DATE);
             hwm :: NN]
            [known = dom birthday;
             !i j :: 1 .. hwm. ~(i = j) ==> ~(names ^^ i = names ^^ j);
             known = {names ^^ i | i :: (1 .. hwm)};
             !i :: 1 .. hwm. birthday ^^ (names ^^ i) = dates ^^ i] /\
            SCHEMA
            [names :: (NN_1 --> NAME);dates :: (NN_1 --> DATE);
             hwm :: NN;names' :: (NN_1 --> NAME);
             dates' :: (NN_1 --> DATE);hwm' :: NN;name? :: NAME;
             date? :: DATE]
            [!i j :: 1 .. hwm. ~(i = j) ==> ~(names ^^ i = names ^^ j);
             !i j :: 1 .. hwm'. ~(i = j) ==> ~(names' ^^ i = names' ^^ j);
             !i :: 1 .. hwm. ~(name? = names ^^ i);hwm' = hwm + 1;
             names' = names (+) {hwm' |-> name?};
             dates' = dates (+) {hwm' |-> date?}] ==>
            (?known' birthday'.
              (known' :: (P NAME) /\ birthday' :: (NAME -+> DATE)) /\
              known' :: (P NAME) /\ birthday' :: (NAME -+> DATE) /\
              (known' = dom birthday') /\
              SCHEMA
              [known' :: (P NAME);birthday' :: (NAME -+> DATE);
               names' :: (NN_1 --> NAME);dates' :: (NN_1 --> DATE);
               hwm' :: NN]
              [known' = dom birthday';
               !i j :: 1 .. hwm'. ~(i = j) ==> ~(names' ^^ i = names' ^^ j);
               known' = {names' ^^ i | i :: (1 .. hwm')};
               !i :: 1 .. hwm'. birthday' ^^ (names' ^^ i) = dates' ^^ i] /\
              SCHEMA
              [known :: (P NAME);birthday :: (NAME -+> DATE);
               known' :: (P NAME);birthday' :: (NAME -+> DATE);
               name? :: NAME;date? :: DATE]
              [known = dom birthday;known' = dom birthday';
               ~name? IN known;
               birthday' = birthday UNION {name? |-> date?}])))))
```

As a final example, here is the "sufficient condition" for the correct implementation of the sequential composition of `AddBirthday` and `FindBirthday` that is described in general terms on page 134 of ZRM. This is much easier to prove than either `AbsThm1` or `AbsThm2`. The sufficient condition for the composition of `AddBirthday1` and `FindBirthday1` is also easy to prove. Note that `show_schema true` is still in force.

```
#prove_theorem                                                                26
# (`AddFindSeq`,
#  "FORALL
#    [BirthdayBook'']
#    ((EXISTS[AddBirthday](theta BirthdayBook' = theta BirthdayBook''))
#     ==>
#     (EXISTS[FindBirthday](theta BirthdayBook = theta BirthdayBook'')))",
#  < omitted tactic > );;

AddFindSeq =
|- !known'' birthday''.
    known'' :: (P NAME) /\ birthday'' :: (NAME -+> DATE) ==>
    known'' :: (P NAME) /\
    birthday'' :: (NAME -+> DATE) /\
    (known'' = dom birthday'') ==>
    (?known birthday known' birthday' name? date?.
      (known :: (P NAME) /\ birthday :: (NAME -+> DATE) /\
       known' :: (P NAME) /\ birthday' :: (NAME -+> DATE) /\
       name? :: NAME /\ date? :: DATE) /\ known :: (P NAME) /\
      birthday :: (NAME -+> DATE) /\ known' :: (P NAME) /\
      birthday' :: (NAME -+> DATE) /\ name? :: NAME /\
      date? :: DATE /\ (known = dom birthday) /\
      (known' = dom birthday') /\ ~name? IN known /\
      (birthday' = birthday UNION name? |-> date?) /\
      ((`birthday`,birthday'),`known`,known' =
       (`birthday`,birthday''),`known`,known'')) ==>
    (?known birthday known' birthday' name? date!.
      (known :: (P NAME) /\ birthday :: (NAME -+> DATE) /\
       known' :: (P NAME) /\ birthday' :: (NAME -+> DATE) /\
       name? :: NAME /\ date! :: DATE) /\ known :: (P NAME) /\
      birthday :: (NAME -+> DATE) /\ known' :: (P NAME) /\
      birthday' :: (NAME -+> DATE) /\ name? :: NAME /\
      date! :: DATE /\ (known = dom birthday) /\
      (known' = dom birthday') /\
      ((`birthday`,birthday'),`known`,known' =
       (`birthday`,birthday),`known`,known) /\
      name? :: known /\
      (date! = birthday ^^ name?) /\
      ((`birthday`,birthday),`known`,known =
       (`birthday`,birthday''),`known`,known''))
```

The implementation of bindings and their extraction (i.e. θ in Z and `theta` in HOL) is discussed in section 3.1.

3 How Z is supported in HOL

The support for Z illustrated in the previous section has two parts: (i) input and output procedures to handle the various schema operations and to manage schema and variable names, and (ii) the HOL theory `Z` that implements the Z operators and mathematical toolkit.

3.1 Inputting and outputting schemas

When a `HOL` quotation of the form `"···"` is read a number of transformations are performed. The most important of these are listed below.

1. Variables that have previously been declared as schema names with the ML function `declare` are replaced by their semantic representation, which is a term of the form `SCHEMA [···] [···]`.

   Decorated (e.g. dashed) schema names are expanded to the appropriately decorated semantic representations (see box 24).

2. Applications $s\ x$, where s has previously been declared in a schema as a set representing a Z function, are expanded to s`^^`x.

3. Terms of the form:

   (a) `pre (SCHEMA [···] [···])`
   (b) `(SCHEMA [···] [···]) SEQ (SCHEMA [···] [···])`
   (c) `(SCHEMA [···] [···]) AND (SCHEMA [···] [···])`
   (d) `(SCHEMA [···] [···]) OR (SCHEMA [···] [···])`
   (e) `(SCHEMA [···] [···]) IMPLIES (SCHEMA [···] [···])`
   (f) `(SCHEMA [···] [···]) HIDE (`$x_1, \cdots, x_n$`)`
   (g) `SCHEMA_FORALL (SCHEMA [···] [···]) (SCHEMA [···] [···])`
   (h) `SCHEMA_EXISTS (SCHEMA [···] [···]) (SCHEMA [···] [···])`

   are expanded out to the appropriate semantic representation of the form `SCHEMA [···] [···]`.

   The input and result of the expansion is saved in a global data structure, so that it can be inverted when schemas are printed.

4. Terms of the form:

   (a) `FORALL (SCHEMA [···] [···])` P
   (b) `FORALL (`x`::`S`)` P
   (c) `EXISTS (SCHEMA [···] [···])` P
   (d) `EXISTS (`x`::`S`)` P

   are expanded out to terms representing the appropriate quantifications of the predicate P. The input and result of the expansion is saved in a global data structure, so it that can be inverted when schemas are printed.

   Quantifications of the form `Q [`v_1`;···;`v_n`]` P are converted into iterated quantifications `Q` v_1 `(Q` v_2 `( ··· (Q` v_n P`) ··· ))`.

5. `sig` (`SCHEMA` $[x_1 : S_1;\ \ldots;\ x_n : S_n]$ $[\cdots]$) is expanded to the type membership statement:

   x_1 `IN` S_1 `/\` x_2 `IN` S_2 `/\ ... /\` x_m `IN` S_m

6. `theta`(`SCHEMA` $[x_1 : S_1;\ \ldots;\ x_n : S_n]$ $[\cdots]$) is expanded to n-tuples of pairs of the form:

   $$((\text{`}x_{\sigma_1}\text{`},\ x_{\sigma_1}),\ \ldots\ ,\ (\text{`}x_{\sigma_n}\text{`},\ x_{\sigma_n}))$$

   which represent *bindings* in Z. The sequence $x_{\sigma_1} \ldots x_{\sigma_n}$ is a canonical reordering of $x_1 \ldots x_n$, which ensures that equivalent bindings (i.e. ones that are equal up to reordering of components) are translated to the same HOL term.

7. `DELTA` $\mathcal{S}$ is translated to $\mathcal{S}$ `AND` $\mathcal{S}'$. This implements the Z definition:

   $$\Delta\mathcal{S}\ \widehat{=}\ \mathcal{S} \wedge \mathcal{S}'$$

8. `XI` $\mathcal{S}$ is translated to `SCHEMA[DELTA` $\mathcal{S}$`][theta`$\mathcal{S}'$ `=` `theta`$\mathcal{S}$`]`. This implements the Z definition:

   $$\Xi\mathcal{S}\ \widehat{=}\ [\Delta\mathcal{S} \mid \theta\mathcal{S}' = \theta\mathcal{S}]$$

When a term is output by HOL, any schema representations that have previously been given a name are replaced by the name. In addition, any representation resulting from of one of the expansions in 3 – 8 above is replaced by the input to the expansion. For example, if S_1 `SEQ` S_2 expands to S_3 (3b above), then S_3 will be output as S_1 `SEQ` S_2. This process is applied recursively.

The method of fully expanding out schemas into their semantic representations works well for small examples like the birthday book. The interactions shown in the boxed sections mostly happen instantaneously (though the proofs of `AbsThm1` and `AbsThm2` take a few seconds to run). However, the underlying terms can get large and it is possible that 'industrial scale' specifications might slow down HOL unacceptably. If this were the case then abbreviatory definitions could be used to prevent terms getting too large. Such abbreviations could be introduced automatically (e.g. by the ML function `declare`). Fortunately performance has been adequate so far and such measures have not been felt necessary.

3.2 Shallow versus deep embedding

Shallow embedding can be contrasted with *deep embedding* [2, 17] in which both the syntax and semantics of the embedded language are formalized inside the host logic. With shallow embedding the mapping from language constructs to their semantic representations is part of the metalanguage; with deep embedding it is part of the object language theory. It is usually more work to support a language by deep embedding, but it is necessary if one wants to prove theorems about the language rather than just reason in it.

Since the schema operations are 'macro expanded' away, it is not possible to state general theorems about them. For example, it is impossible to express the fact that schema conjunction is commutative. For any particular schema one can prove the instance of the fact, but such proofs have to be repeated for each separate instance (though they can be performed automatically with a suitable derived rule).

A weakness of shallow embedding is that the operations that are computed outside the logic are not subject to the same 'quality control' as operations specified in the logic, because errors in the definition of operators reside in program code not in logical formulae. Generally the latter are easier to inspect for correctness than the former. For example, if the macro expansion of `SEQ` contained a bug (e.g. dashed variables were sometimes invalidly captured by quantifiers) then this might only manifest itself in the wrong schema expansion being computed. If in addition the output routines 'inverted' the bug the user might be unaware that the wrong semantic representation was being manipulated. A deep embedding allows meta-theorems to be proved (e.g. the associativity of schema sequencing) that can serve to partially validate the definitions of the operators.

A deep embedding of Z in HOL is possible. It could, for example, be based on the metatheory presented in the Z base standard [4]. The resulting theory would be complex and probably hard to apply to particular examples like the birthday book. However, it would be suitable for testing out the meta-theory of Z and verifying general properties of it (e.g. see the work by Maharaj briefly discussed below).

The distinction between "shallow" and "deep" is not always sharp. For example, ProofPower provides a much 'deeper' embedding than the one described here (all the Z operators are defined in HOL) but there is no single semantic function defined in the the logic that maps Z syntax into its meaning. ProofPower's embedding is not 'deep enough' to allow facts like the commutativity of schema conjunction to be proved.

The lightweight shallow embedding illustrated here puts relatively few obstacles in the way of using all the power of HOL to reason about particular Z specifications, but it is useless for verifying properties of Z itself.

3.3 Z's operators in HOL

The operators used in the birthday book are included in those shown in the table below. The ASCII versions of the Z operators are based on those used by the *f*UZZ and ZTC type-checking tools [28, 11].

Z operator	HOL notation	Meaning
:	`::`	Membership declaration
dom	`dom`	Domain
$\mapsto$	`|->`	Maplet
$\mathbb{P}$	`P`	Power set
$\times$	`><`	Cartesian product
$\leftrightarrow$	`<->`	Binary relations
$\nrightarrow$	`-+>`	Partial functions
$\rightarrow$	`-->`	Total functions
	`^^`	Application of Z functions
$\lhd$	`<+`	Domain anti-restriction
$\oplus$	`(+)`	Relational overriding
$\mathbb{N}$	`NN`	Natural numbers
$\mathbb{N}_1$	`NN_1`	Strictly positive integers
..	`..`	Number range

The operators in this table are defined (in HOL notation) by:

```
|- (CONJL[] = T) /\ (!b bl. CONJL(CONS b bl) = b /\ CONJL bl)
|- SCHEMA decs body = CONJL decs /\ CONJL body
|- x :: s = x IN s
|- x |-> y = x,y
|- dom R = {x | ?y. (x |-> y) IN R}
|- P X = {Y | Y SUBSET X}
|- X >< Y = {(x,y) | x IN X /\ y IN Y}
|- X <-> Y = P(X >< Y)
|- X -+> Y = {f | f IN (X <-> Y) /\ (!x y1 y2.
           (x |-> y1) IN f /\ (x |-> y2) IN f ==> (y1 = y2))}
|- X --> Y = {f | f IN (X -+> Y) /\ (dom f = X)}
|- f^^x = @y. (x,y) IN f
|- S <+ R = {x |-> y | ~x IN S /\ (x |-> y) IN R}
|- f (+) g = ((dom g) <+ f) UNION g
|- NN = {n | n >= 0}
|- NN_1 = {n | n > 0}
|- m .. n = {i | m <= i /\ i <= n}
```

The HOL function CONJL conjoins a list of predicates. The set application operator ^^ is defined using the HOL choice operator @, which is Hilbert's ε-symbol. The term $@x\,.\,P[x]$ denotes some value, a say, such that $P[a]$ is true. @ is related to the Z μ-operator, but unlike μ does not require $P[x]$ to be satisfied by a unique value. The conventional logical symbol corresponding to μ in Z is the unique existence operator ι. If no a exists such that $P[a]$ is true, then $@x\,.\,P[x]$ denotes an arbitrary value. It would be possible to choose this arbitrary value in a canonical way, but its underlying HOL type must be the same as that of x in $P[x]$.

The theory Z also contains HOL theorems about the Z operators. The ones used in proving the theorems in section 2 are listed below:

```
|-  dom(X UNION Y) = (dom X) UNION (dom Y)
|-  dom{x |-> y} = {x}
|-  x IN (dom{x |-> y})
|-  f IN (X -+> Y) /\ x IN (dom f) ==> x IN X
|-  f IN (X -+> Y) ==> (dom f) IN (P X)
|-  x IN X /\ y IN Y ==> {x |-> y} IN (X -+> Y)
|-  f IN (X -+> Y) /\ x IN X /\ y IN Y /\ ~x IN (dom f) ==>
    (f UNION {x |-> y}) IN (X -+> Y)
|-  f IN (X --> Y) ==> (dom f = X)
|-  f IN (X --> Y) ==> f IN (X -+> Y)
|-  ~(x1 = x2) ==> ((X UNION {x1 |-> v}) ^^ x2 = X ^^ x2)
|-  ~x IN (dom X) ==> ((X UNION {x |-> v}) ^^ x = v)
|-  {x |-> v} ^^ x = v
|-  f IN (X -+> Y) /\ x IN (dom f) ==>
    (!y. (f ^^ x = y) = (x,y) IN f)
|-  f IN (X -+> Y) ==> (!x. x IN (dom f) = (x,f ^^ x) IN f)
|-  f IN (X --> Y) /\ g IN (X --> Y) ==>
    (f (+) g) IN (X --> Y)
```

```
|-  f IN (X -+> Y) /\ g IN (X -+> Y) ==>
    (f (+) g) IN (X -+> Y)
|-  f IN (X -+> Y) /\ g IN (X -+> Y) ==>
    (dom(f (+) g) = (dom f) UNION (dom g))
|-  f IN (X -+> Y) /\ g IN (X -+> Y) /\
    x IN ((dom f) DIFF (dom g)) ==> ((f (+) g) ^^ x = f ^^ x)
|-  f IN (X -+> Y) /\ g IN (X -+> Y) /\ x IN (dom g) ==>
    ((f (+) g) ^^ x = g ^^ x)
|-  n IN NN
|-  (n + 1) IN NN_1
|-  f IN (NN_1 --> X) /\ v IN X ==>
    ((f (+) {(n + 1) |-> v}) ^^ (n + 1) = v)
|-  1 .. (n + 1) = (1 .. n) UNION ((n + 1) .. (n + 1))
|-  x IN (n .. n) = (x = n)
|-  x IN (m .. n) = m <= x /\ x <= n
|-  f IN (NN_1 -+> X) /\ x IN X ==>
    (f (+) {(n + 1) |-> x}) IN (NN_1 -+> X)
|-  NN_1 UNION {n + 1} = NN_1
|-  f IN (NN_1 --> X) /\ x IN X ==>
    (f (+) {(n + 1) |-> x}) IN (NN_1 --> X)
```

4 Other related work

ICL's ProofPower [12] is an 'LCF-like' system that supports the same version of higher order logic as HOL, but provides different theorem proving infrastructure. ICL used the original HOL for many years and have built on this experience in designing ProofPower. Although the theorem proving tools are not identical to HOL's, they are similar in spirit but aim to be more powerful. Some of the features provided have been designed with the needs of Z in mind. ProofPower supports Z via a much 'deeper' embedding than the one presented here. The main difference in their approach is that schemas are treated more like set abstractions (yielding a set of bindings), and schema operations are then operators over sets of bindings. Schema references as predicates are treated as abbreviations for membership statements, e.g. a reference S abbreviates $\theta S \in \hat{S}$, where $\hat{S}$ is the representation of schema S. ProofPower has been used for internal applications at ICL and is currently undergoing evaluation in a small group of companies and academic institutions. ProofPower proofs in Z are intended to be conducted with all visible subgoals in Z notation. The language supported is intended to be compatible with the proposed 'ISO standard Z' [4] rather than the Z described in ZRM. ProofPower has a polished interface that understands Z's special symbols. It has more Z-specific proof infrastructure and better coverage of the Z notation than the implementation described here. More information is obtainable from the ProofPower server by sending an email message to: `ProofPower-server@win.icl.co.uk`.

A relatively deep embedding of Z in the type theory UTT (Unifying Theory of dependent Types) has been undertaken by Savi Maharaj [15]. A method of representing Z schemas in UTT (independently of the Z core language) was developed. The LEGO proof-checker has been used to prove several theorems that are intended to support reasoning about Z-like specifications at the schema level. For example, introduction and elimination rules for schema conjunction have been verified. Earlier work [14] considered ways of encoding the core language of Z in type theory.

The Z/EVES project at Odyssey Research Associates (ORA) [24, 25] investigated the feasibility of using EVES, a theorem prover for ZF set theory, as a proof tool for Z. A standard example from the security community ("The Low Water Mark") was used as a case study. This was specified in Z and a proof of a non-interference security property performed. Most of the Mathematical Toolkit in ZRM was expressed in EVES and the laws proved and installed as rewrites. EVES is much more automatic than HOL and thus an embedding of Z into EVES might provide proof support with less user interaction. However, the feasibility study was not followed by any tool development.

The Balzac project at Imperial Software Technology (IST) is building editing, typesetting, type-checking and proof support for Z [10]; it is funded by contracts from CESG (a UK Government organization). Balzac is a tactic based system in which tactics rather than rules are primitive; the tactic writing language is a secure form of Lisp. The user interface is powerful and sophisticated. The version of Z supported is non-standard (e.g. basic types are assumed non-empty). Balzac has been used to produce specifications consisting of 500–1000 pages (about two and a half schemas a page) and has been used to carry out proofs about these which involve a significant proportion of the schemas. IST sell a commercial variant of Balzac called Zola. [5]

Another approach to providing proof support for Z is to encode a deductive system for it in a generic theorem prover. An example of this is the encoding of $\mathcal{W}$, a logic for Z [30], in the 2OBJ metalogical theorem prover [16]. This approach differs from semantic embedding in that the semantics of Z is not represented in the host logic; instead proof rules are encoded directly. This has the advantage that complex rules can be immediately implemented as primitives. With semantic embedding such rules have to be embodied in procedures (e.g. ML functions) that perform inferences in an underlying logic. A single step in a logic designed specially for Z (e.g. $\mathcal{W}$) might correspond to many steps in a more primitive logic (e.g. higher order logic). For example, the proof of `AbsThm2` above consists of over five thousand primitive HOL inferences. Currently the encoding of $\mathcal{W}$ in 2OBJ is very inefficient and so it is not yet usable in practice.

A more practical approach, not based on semantic embedding, was adopted by the *zed***B** tool [19, 20]. This was based on the B-Tool (which is a general-purpose symbolic manipulator not incorporating any particular logic) and enabled calculation and verification of properties of Z specifications. Facilities for schema viewing, expansions, precondition calculation, discharging of initialization and other syntactic and semantic operations were included. Unfortunately only a prototype tool which is not generally available has been produced.

Many Z tools are under development; current information on the availability of Z tools may be gleaned from the monthly message issued on the electronic newsgroup `comp.specification.z` and the associated ZFORUM mailing list on Z [3].

[5] The example from the report "A Simple Demonstration of Balzac" [6] has been done in HOL; see `contrib/Z/TelephoneBook.ml` distributed with HOL88 Version 2.02.

5 Conclusions and future work

It turned out to be easier than expected to provide basic support for a significant subset of Z in HOL. At the time of writing only a few case studies have been conducted (the birthday book is the largest), so it is hard to evaluate the success or otherwise of the approach. Only a fragment of Z is currently supported, but no major difficulties are anticipated in increasing the coverage to most of the rest of the Z toolkit. Other more complicated features of Z such as schemas as types, true generic definitions and the complete facilities for free type definitions may require further thought and/or compromises. Our plan is to approach this via more case studies. Examples being considered include the simple checkpointing scheme in ZRM, the "Word-For-Word" example in the textbook *An Introduction to Formal Specification and Z* [23], an ML pattern matching refinement case study [13] and the "The Low Water Mark" [24].

A disadvantage of our approach is that users need to be proficient with HOL before they can attempt proofs in Z. Learning HOL is quite time consuming; at least one week of full-time training is needed to get started effectively. In particular, HOL tactics must be mastered. However, extensive training materials for HOL are available and there is a relatively large international user community.

In the future it is hoped to provide support for interfacing with the widely available LaTeX document preparation system for compatibility with other Z-processing tools, such as *f*UZZ (whose LaTeX style option was used to prepare this paper) and the public domain ZTC [11]. HOL's parser and pretty printer libraries should make this relatively straightforward.

Acknowledgements

We are grateful to Jim Grundy, Terry Ireland, Roger Jones, Savitri Maharaj, Tom Melham and Mark Saaltink for reading a draft of this paper and making numerous suggestions for its improvement. Will Harwood provided information on Balzac and its use. Mike Spivey provided the Birthday Book example and the *f*UZZ tool. Jonathan Bowen is funded by the UK Science and Engineering Research Council (SERC) on grant no. GR/J15186.

References

[1] Andrews PD. An Introduction to Mathematical Logic and Type Theory: To Truth through Proof. Computer Science and Applied Mathematics Series. Academic Press, 1986.

[2] Boulton RJ, Gordon AD, Harrison JR, Herbert JMJ, Van Tassel J. Experience with embedding hardware description languages in HOL. In Stavridou V, Melham TF, Boute RT (eds), Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP TC10/WG 10.2 International Conference, IFIP Transactions A-10, pp 129–156. North-Holland, 1992.

[3] Bowen JP. Comp.specification.z and Z FORUM frequently asked questions. In Bowen JP, Nicholls JE (eds), Z User Workshop, London 1992, Workshops in Computing, pp 342–347. Springer-Verlag, 1993.

[4] Brien SM, Nicholls JE. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, UK, 1992. Accepted for ISO standardization, ISO/IEC JTC1/SC22.

[5] Church A. A formulation of the simple theory of types. The Journal of Symbolic Logic, 5:56–68, 1940.

[6] Collinson R. A simple demonstration of Balzac. Technical report, GCHQ, Fiddlers Green Lane, Cheltenham, Gloucestershire, 1992.

[7] Diller A. Z: An Introduction to Formal Methods. Wiley, 1990.

[8] Gordon MJC, Melham TF (eds). Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic. Cambridge University Press, 1993.

[9] Gordon MJC, Milner R, Wadsworth CP. Edinburgh LCF: A Mechanised Logic of Computation, vol 78 of Lecture Notes in Computer Science. Springer-Verlag, 1979.

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

[11] Jia X. ZTC: A Type Checker for Z User's Guide. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604, U.S.A. (E-mail: `jia@cs.depaul.edu`), 1994.

[12] Jones RB. ICL ProofPower. BCS FACS FACTS, 1(1):10–13, 1992. Series III.

[13] Macdonald R, Randell GP, Sennett CT. Pattern matching in ML: A case study in refinement. Report No. 89004, RSRE (now DRA), Defence Research Agency, St. Andrews Road, Malvern, Worcs WR14 3PS, UK, 1989.

[14] Maharaj S. Implementing Z in LEGO. Master's thesis, University of Edinburgh, 1990.

[15] Maharaj S. Encoding Z schemas in type theory. In Geuves H (ed), Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs, pp 209–218, 1993. Distributed electronically.

[16] Martin A. Encoding $\mathcal{W}$: A logic for Z in 2OBJ. In Woodcock JCP, Larsen PG (eds), FME'93: Industrial-Strength Formal Methods, vol 670 of Lecture Notes in Computer Science, pp 462–481. Springer-Verlag, 1993.

[17] Melham T. Using recursive types to reason about hardware in Higher Order Logic. In Milne GJ (ed), The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG10.2 Working Conference, pp 27–50. North-Holland, 1988.

[18] Milner R, Tofte M, Harper R. The Definition of Standard ML. The MIT Press, 1990.

[19] Neilson D. Machine support for Z: the zedB tool. In Nicholls JE (ed), Z User Workshop, Oxford 1990, Workshops in Computing, pp 105–128. Springer-Verlag, 1991.

[20] Neilson D, Prasad D. zedB: A proof tool for Z built on B. In Nicholls JE (ed), Z User Workshop, York 1991, Workshops in Computing, pp 243–258. Springer-Verlag, 1992.

[21] Paulson LC. Logic and Computation: Interactive Proof with Cambridge LCF, vol 2 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[22] Paulson LC. A fixedpoint approach to implementing (co-)inductive definitions. Technical report, University of Cambridge, Computer Laboratory, 1993. Draft.

[23] Potter BF, Sinclair JE, Till D. An Introduction to Formal Specification and Z. Prentice Hall International Series in Computer Science, 1990.

[24] Saaltink M. Z and EVES. Technical Report TR-91-5449-02, Odyyssey Research Associates, 265 Carling Avenue, Suite 506, Ottawa, Ontario K1S 2E1, Canada, 1991.

[25] Saaltink M. Z and Eves. In Nicholls JE (ed), Z User Workshop, York 1991, Workshops in Computing, pp 223–242. Springer-Verlag, 1992.

[26] Smith A. On recursive free types in Z. In Nicholls JE (ed), Z User Workshop, York 1991, Workshops in Computing, pp 3–39. Springer-Verlag, 1992.

[27] Spivey JM. An introduction to Z and formal specifications. IEE/BCS Software Engineering Journal, 4(1):40–50, 1989.

[28] Spivey JM. The fUZZ Manual. Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 2nd edition, 1992.

[29] Spivey JM. The Z Notation: A Reference Manual. Prentice Hall International Series in Computer Science, 2nd edition, 1992. (1st edition, 1989).

[30] Woodcock JCP, Brien SM. $\mathcal{W}$: A logic for Z. In Nicholls JE (ed), Z User Workshop, York 1991, Workshops in Computing, pp 77–96. Springer-Verlag, 1992.