2.5 The Z Notation

[Reference: M. Spivey: The Z Notation, Prentice Hall]
Properties of Z

- Is a very expressive formal language (well-defined syntax and semantics).

- Based on first-order logic with equality and typed set-theory.

- Has a mathematical toolkit: a library of mathematical definitions and abstract data-types (sets, lists, bags, ...).

- Supports the structured modeling of a system, both static and dynamic:
  - modeling/specification of data of the system,
  - functional description of the system (state transitions).

- Is supported by several tools and systems.
Z and other Formal Languages/Methods

- A number of successfully employed Formal Languages are based on first order logic with type-theory, e.g.
  - VDM ("Vienna Development Method", 80's),
  - B (applied extensively in France)

- Other formal languages:
  - Equational logic or Horn logic (in algebraic specifications),
  - Higher-order logic (HOL).

- Z:
  - Applied successfully since 1989 (Oxford University Computing Laboratory), e.g. British government requires Z-specifications for safety-critical systems.
  - Is an ISO standard.
The Birthday Book Example

- A system which records people’s birthdays
- System issues a reminder when the birthday comes around
The Birthday Book Example - Schema

BirthdayBook

known: \( P \ NAME \)
birthday: \( NAME \rightarrow DATE \)

known = dom birthday

Invariants (variable relationships)

One instance satisfying the invariant:

\[ \text{known} = \{ \text{John, Mike, Susan} \} \]

\[ \text{birthday} = \{ \text{John} \rightarrow 25. \text{ Mar}, \]
\[ \text{Mike} \rightarrow 20. \text{ Dec}, \]
\[ \text{Susan} \rightarrow 20. \text{ Dec.} \} \]
Z syntax overview

- A specification in Z is presented as a collection of schemas.

- Each schema consists of:
  - Schema name
  - Variable declaration (static description part)
    - Identification of variables
  - Invariants (dynamic part)
    - Definition of operations
    - Input/Output relation
    - State changes

- Z uses only rigorous mathematical notations from (typed) set theory and first order logic.
The Birthday Book Example – Details to BirthdayBook

BirthdayBook

\[ \text{known: } \mathcal{P} \, \text{NAME} \]
\[ \text{birthday: } \text{NAME} \rightarrow \text{DATE} \]

\[ \text{known} = \text{dom} \, \text{birthday} \]

- \( \mathcal{P} \): power set
- \( \text{dom} \rightarrow \text{domain} \)
  - The set \( \text{known} \) is the same as the domain of the function \( \text{birthday} \) – the set of names to which it can be validly applied.
  - \textit{Facts about dom} (examples of the laws obeyed by mathematical data types):
    - \( \text{dom}(f \cup g) = (\text{dom } f) \cup (\text{dom } g) \)
    - \( \text{dom} \{a \rightarrow b\} = \{a\} \)
The Birthday Book Example - Operation

AddBirthday

\[ \Delta \text{BirthdayBook} \]

\begin{align*}
\text{name?} & : \text{NAME} \\
\text{date?} & : \text{DATE} \\
\end{align*}

\[\text{name?} \notin \text{known}\]

\[\text{birthday}' = \text{birthday} \cup \{\text{name?} \rightarrow \text{date?}\}\]

- \(\Delta \rightarrow\) introduces a schema describing a change
- Introduces variables: known, birthday (state before change)
- known', birthday' (state after the change)
- ? \rightarrow Input variables
- ! \rightarrow output variables
- Constraint: All variables with the same name have the same type
A simple proof of a system property

- Expectation: set of known names will be augmented with the new name

→ Property: known' = known ∪ {name?}

- Proof:

  known' = dom birthday'

  = dom (birthday ∪ {name? → date?}) [spec. of AddBirthday]

  = dom birthday ∪ dom {name? → date?} [fact about ‘dom’]

  = dom birthday ∪ {name?} [fact about ‘dom’]

  = known ∪ {name?} □ [invariant before]

- Proof based on mathematical laws and axioms
Find operation

- **FindBirthday**

  ▢ ∀BirthdayBook
  
  name?: NAME
  
  date!: DATE

  name? ∈ known
  
  date! = birthday (name?)

- Finds birthday (date) for a given name
- ∀ → introduces a schema without a change
- Constraint: all variables with the same name are of the same type
Remind operation

Remind

Existential BirthdayBook

- \( \text{today?} : \text{DATE} \)
- \( \text{cards!} : \text{P} \text{ NAME} \)

\[ \text{cards!} = \{ n : \text{known} \mid \text{birthday} (n) = \text{today?} \} \]

- Finds people who have birthday on a given date (to send them birthday cards)
- No precondition!
Initial state

InitBirthdayBook

BirthdayBook

known = ∅
Strengthen the specification (1/2)

- What is the problem with this spec?
  - Assumes everything works according to plan
  - No provision for mistake
    - What happens if user attempts to add a birthday for someone already known to the system?
    - Or if we try to find a birthday of someone not known to the system?
      - Should the system ignore such behavior?
      - Should the system break down by displaying rubbish?
- Does this mean we should write a brand new spec?
**Strengthen the specification (2/2)**

- How to strengthen the specification?
  - Combine separate schema
    - Add an extra output (or report) to each operation on the system
    - Outputs can be:
      - Ok
      - Already-known
      - Not-known
Success Operation

Success

\[ \text{result!} : \text{REPORT} \]

\[ \text{result!} = \text{ok} \]

- To combine schema use conjunction or disjunction:
  - \textbf{AddBirthday} \land \textbf{Success} is a schema which produces an entry in the birthday book (for correct inputs) and reports success
  - The result is an operation which, for correct input, both acts as described by \textit{AddBirthday} and produces the result \textit{ok}. 
AlreadyKnown Operation

- Define a schema for each possible error in the input
- \( R\text{AddBirthday} \triangleq (\text{AddBirthday} \land \text{Success}) \lor \text{AlreadyKnown} \)
A robust version of \( \text{AddBirthday} \)

\[
\text{RAddBirthday} \\
\triangle \text{BirthdayBook} \\
\text{name?} : \text{NAME} \\
\text{date?} : \text{DATE} \\
\text{result!} : \text{REPORT} \\
\]

\[
(\text{name?} \notin \text{known} \land \\
\text{birthday}' = \text{birthday} \cup \{\text{name?} \rightarrow \text{date?}\} \land \\
\text{result!} = \text{ok}) \lor \\
(\text{name?} \in \text{known} \land \\
\text{birthday}' = \text{birthday} \land \\
\text{result!} = \text{already\_known})
\]
A robust version of FindBirthday

\[ \text{RFindBirthday} \overset{=} {=} (\text{FindBirthday} \land \text{Success}) \lor \text{NotKnown} \]

\[ \text{NotKnown} \]

\[ \exists \text{BirthdayBook} \]
\[ \text{name?} : \text{NAME} \]
\[ \text{result!} : \text{REPORT} \]

\[ \text{name?} \notin \text{known} \]
\[ \text{result!} = \text{not\_known} \]
From specification to design

- Thus far:
  - How to specify software modules using Z
  - How to compose larger specifications using schema calculus

- Need to know how Z can be used to document the design of a program
- Need to describe data and operation refinement
Schema Refinement

- Stepwise refinement to reach an implementation:
  - Data refinement
    - Describe the concrete data structures which the program will use represented by the abstract data type of the specification
  - Operation refinement
    - Derive the description of the operations in terms of the concrete data structure
Refinement Process

- $S \subseteq S' \colon \text{"S‘ refines S"}

Increasing abstraction

$S \subseteq S_1 \subseteq S_2 \subseteq S_3 \subseteq \ldots \subseteq S_n \subseteq S'$

Increasing implementability
Data Refinement

- Use a pair of arrays to implement the abstract data types NAME and DATE
  - names: array [1…] of NAME;
  - dates: array [1…] of DATE;

- Mathematical model of the arrays by mathematical functions
  - names: \( \mathbb{N}_1 \rightarrow \text{NAME} \)
  - dates: \( \mathbb{N}_1 \rightarrow \text{DATE} \)
## Implementation of Birthday Book

<table>
<thead>
<tr>
<th>BirthdayBook1</th>
</tr>
</thead>
<tbody>
<tr>
<td>names: ( N_1 \rightarrow \text{NAME} )</td>
</tr>
<tr>
<td>dates: ( N_1 \rightarrow \text{DATE} )</td>
</tr>
<tr>
<td>hwm: ( \mathbb{N} )</td>
</tr>
</tbody>
</table>

\[
\forall i, j : 1 .. \text{hwm} \bullet \\
\quad i \neq j \rightarrow \text{names}(i) \neq \text{names}(j)
\]

- \( hwm \): (high water mark) shows how much of the array is in use
- Invariant: no repetition among the names (i.e. names are unique)
Mapping from specification to implementation

- **Abs** defines the abstraction relation between the abstract state space *BirthdayBook* and the concrete state space *BirthdayBook1*

\[
\text{known} = \{i : 1..\text{hwm} \bullet \text{names}(i)\}
\]
\[
\forall i : 1..\text{hwm} \bullet \text{birthday}(\text{names}(i)) = \text{dates}(i)
\]
Implementation of AddBirthday

---

AddBirthday1

\[ \Delta \text{BirthdayBook1} \]

\text{name? : NAME}

\text{date? : DATE}

---

\[ \forall i : 1 \ldots \text{hwm} \bullet \text{name?} \neq \text{names}(i) \]

\[ \text{hwm}' = \text{hwm} + 1 \]

\[ \text{names}' = \text{names} \oplus \{\text{hwm}' \rightarrow \text{name?}\} \]

\[ \text{dates}' = \text{dates} \oplus \{\text{hwm}' \rightarrow \text{date?}\} \]

\[ \text{names} \oplus \{\text{hwm}' \rightarrow \text{name?}\} \]

operation takes all values of names and does not change them except at argument hwm’ where it takes value of name?
Proving correct refinement

- Operation Refinement allows to prove that an operation is a correct implementation of another operation with the same state space.

- Concrete and abstract operations can differ in two ways:
  1. The precondition of the abstract operation can be more liberal than the precondition of the concrete operation.
  2. The concrete operation can be more deterministic than the abstract operation.
Properties of correct refinement

1. The concrete operation must terminate whenever the abstract operation terminates.
   - Birthday book: Whenever AddBirthday is legal in some abstract state, the implementation AddBirthday1 is legal in any corresponding concrete state.
   - Precondition (AddBirthday) ⇒ Precondition(AddBirthday1)

2. If the abstract operation terminates, each state that the concrete operation might produce must be one of the states that the abstract operation might produce.
   - Birthday book: The final state which results from AddBirthday1 represents an abstract state which AddBirthday could produce.
   - Precondition(AddBirthday) ∧ AddBirthday1 ⇒ AddBirthday
Proof of Property 1

- Property 1: Precondition (AddBirthday) $\Rightarrow$ Precondition(AddBirthday1)

**to prove:**

name? $\notin$ known  \[Precond. of AddBirthday\] is the only legal state of AddBirthday except for final state]

$\Rightarrow$ known = \{i : 1 .. hwm $\bullet$ names(i)\} \[Definition of Abs\]
$\Rightarrow$ $\forall$ i : 1 .. hwm $\bullet$ name? $\neq$ names(i) $\square$ \[Precondition of AddBirthday1\]

which is the only legal state of AddBirthday1 except for the final state
Proof of Property 2 (1/2)

to prove:

\[ \text{birthday}' = \text{birthday} \cup \{ \text{name}? \rightarrow \text{date}? \} \]  
(Postcondition of AddBirthday holds after applying AddBirthday1)

1. \[ \text{names}'(i) = \text{names}(i) \land \text{dates}'(i) = \text{dates}(i) \]  
   \[ \forall i : 1 \ldots \text{hwm} \]  
   [Def. of \( \oplus \)]

   \[ \Rightarrow \text{birthday}'(\text{names}'(i)) = \text{dates}'(i) = \text{dates}(i) = \text{birthday}(\text{names}(i)) \]  
   [Def. of Abs]

   [1.]

2. \[ \text{birthday}'(\text{names}'(\text{hwm}')) = \text{dates}'(\text{hwm}') \]  
   [Def. of \( \oplus \)]

   \[ \Rightarrow \text{birthday}'(\text{name}?) = \text{birthday}'(\text{names}'(\text{hwm}')) = \text{dates}'(\text{hwm}') = \text{date}? \]  
   [Def. of AddBirthday1]

   [2.]
Proof of Property 2 (2/2)

3. \[ hwm' = hwm + 1 \]

\[ \Rightarrow \text{dom birthday'} = \text{known'} \]

= \{i : 1… hwm' • names'(i)}

= \{i : 1… hwm • names'(i)} ∪

\{names'(hwm')\}

= \{i : 1… hwm • names(i)} ∪

\{name?\}

= known ∪ \{name?\}

= dom birthday ∪ \{name?\} □

[invariant from BirthdayBook]

[Def. of Abs]

[3.]

[Def. of AddBirthday1]

[Def. of Abs]

[invariant from BirthdayBook]
Reminder: AddBirthday1

**AddBirthday1**

\[ \\Delta BirthdayBook1 \]
\[
\text{name? : NAME} \\
\text{date? : DATE}
\]

\[ \forall \ i : 1 \ldots \ hwm \bullet \text{name ?} \neq \text{names(i)} \]
\[
hwm' = hwm + 1 \\
names' = names \oplus \{hwm' \rightarrow \text{name?}\} \\
dates' = dates \oplus \{hwm' \rightarrow \text{date?}\}
\]

- Uses only notation that has a direct counterpart in the programming language
Implementation of 
AddBirthday1

\begin{verbatim}
procedure AddBirthday (name : NAME; date: DATE);
begin
  hwm := hwm + 1;
  names[hwm] := name;
  dates[hwm] := date
end;
\end{verbatim}
Refinement of FindBirthday

FindBirthday1

$\exists$ BirthdayBook1
name? : NAME
date! : DATE

$\exists \; i = 1..hwm \bullet$

name? = names(i) $\land$ date! = dates(i)

- Precondition: name? must appear in names
- Don’t need to check final state (no state change)
- Need to check precondition of FindBirthday1 and correctness of output date!
Abstract Property which must hold

Proof of Precondition

\( \text{name?} \in \text{known} \Rightarrow \exists i = 1 \ldots \text{hwm} \cdot \text{name?} = \text{names}(i) \quad \text{[Def. of Abs]} \)

Proof of Postcondition

\( \exists i = 1 \ldots \text{hwm} \cdot \text{name?} = \text{names}(i) \land \text{date!} = \text{dates}(i) \)

\( \Rightarrow \text{date!} = \text{dates}(i) \quad \text{[Def. of FindBirthday1]} \)
\( = \text{birthday}(\text{names}(i)) \quad \text{[Def. of Abs]} \)
\( = \text{birthday}(\text{name?}) \quad \text{[Def. of FindBirthday1]} \)
procedure FindBirthday (name : NAME; var date : DATE);
    var i : INTEGER;
begin
    i := 1;
    while names[i] ≠ name
        do  i := i + 1;
            date := dates[i]
end;
Refinement of Remind (1/3)

- Problem: cards is a set of names
  - Cannot be directly represented in the programming language
- Solution: define an abstraction relation that is represented by an array and an integer

Remind

\[ \exists \text{BirthdayBook} \]
\[ \text{today?} : \text{DATE} \]
\[ \text{cards!} : \text{P NAME} \]

\[ \text{cards!} = \{ n : \text{known} \mid \text{birthday} (n) = \text{today?} \} \]
AbsCards

cards : P NAME
-cardlist : N₁ → NAME
-ncards : N

cards = \{i : 1.. ncards \cdot cardlist(i)\}
Refinement of Remind (3/3)

- Output: integer $ncards$, array $cardlist$
- The set contains all names in the $names$ array for which the corresponding entry in the $dates$ array is $today$. 
Implementation of Remind1

```pascal
procedure Remind(today : DATE;
    var cardlist : array [1..] of NAME;
    var ncards : INTEGER);

    var j : INTEGER
begin
  ncards := 0;
  j := 0;
  while j < hwm do begin
    j := j + 1;
    if dates [j] = today then begin
      ncards := ncards + 1;
      cardlist[ncards] := names[j]
    end
  end
end;
```
Implementation of the initial state

InitBirthdayBook1

BirthdayBook1

hwm = 0

\[ \text{known} = \{ i : 1..\text{hwm} \cdot \text{names} (i) \} \]
\[ = \{ i : 1..0 \cdot \text{names}(i) \} \]
\[ = \emptyset \]

procedure InitBirthdayBook;
begin
hwm: = 0
end;
Summary

- Direct refinement: One step from specification to implementation
- Abstraction schema: relationship between abstract states and program variables
- Simple example
- More complex examples require several iterative refinement steps