INTRDUCTION TO Z SPECIFICATIONS

Formal Methods

BCS 2213

Semester 2 Session 2013/2014

What is Z

- First off – its pronounced Zed
  - After the Zermelo-Fränkel set theory
- Z is based on the mathematical notation used in axiomatic set theory, lambda calculus and first order predicate logic.
- It has been developed by the Programming Research Group at the Oxford University Computing Laboratory and elsewhere since the middle of 1980.

What is Z

- *Z – is a modelling notation.*
- Can be used to model the behavior of a system.
- Model the system by representing its *state* (a collection of *state variables* and their values) and *operations* that may change the state.
- Z is just a notation, it is not a method.
- Can support many different methods.

Z is….

- Z is not software or executable code.
- Describe "what" the system must do without saying "how" it is to be done.
- Can be used to represent structure of software code: data types, procedures, functions, modules, classes, objects etc.
- Z is suitable for
  - Procedural programming language
  - Functional approach
  - Object-oriented programming. There are some object oriented languages that extend Z (Object-Z, Z++)

Z is…

- Designed for people not machines
- Z Notation is a mixture of boxes, text, Greek letters and invented pictorial symbols.
- Includes two notations
  - Notation to express ordinary discrete mathematics
  - Notation to structure mathematical text (*schema*).
- Problems with Z notation
  - Z notation uses many non-ASCII symbols
  - The specification includes suggestions for rendering the Z notation symbols in ASCII as well as LaTeX.

Z Mathematical tool-kit

- Consist of a small core, supplemented by large collection of useful objects and operators.
- Collection of mathematical theories: definitions and laws concerning objects such as sets, tuples, relations, functions, sequences and their operators.
- Plays the same role as standard library of types and functions in programming.
- Sample of Z Mathematical tool-kit : http://staff.washington.edu/jon/z/toolkit.html

How To Model A System

- Explicitly describes behavior in terms of mathematical types (set, sequences, relations, functions) & defines operations
- Z specification includes
  - types - syntax of object being specified
  - invariants - properties of modeled object
  - pre/post conditions – semantics of operations
- Z decomposes specifications into manageably sized module's, called schemas. Schemas are divided into 3 parts:
  - A State
  - A collection of *state variables* and their values
  - There are also *operations* that can change its state

How To Model Static and Dynamic Aspects

- Static aspects
  - The *states* that a system can occupy.
  - The *invariant relationships* that are maintained as the system moves from state to state.
- Dynamic aspects
  - The *operations* that are possible.
  - The *relationship* between their inputs and outputs.
  - The *changes* of state that happen.

How to Specify Static Aspects?

Use *schemas* - math in a box with a name - to describe the state space (state variables and invariants).

Name of schema

Declaration of state variables.

Invariant relationship between values of the variables

The Birthday Book Example

- This example will allow you to do 3 things:
  - Add a person's name and birthday
  - Store that information
  - Find a birthday by name
  - Find a name by given date of birthday
- One possible state of the system has three people in the set **known**, with their birthdays recorded by the function **birthday**:

$$\text{known} = \{\ \text{John, Mike, Susan}\ \}$$

$$\text{birthday} = \{\ \begin{array}{lcl} \text{John} & \longrightarrow & \text{25-Mar}, \\ \text{Mike} & \longrightarrow & \text{20-Dec}, \\ \text{Susan} & \longrightarrow & \text{20-Dec} \end{array}\ \}$$

Example: Birthday Book

- BirthdayBook schema for recording people's birthdays
  - known: set of names with birthdays recorded
  - birthday: function from names to birthdays
  - Q: What does the **invariant** say?

$$
\begin{array}{|l}
\hline
\textit{BirthdayBook} \\
\hline
\textit{known} : \mathbb{P}\ \textit{NAME} \\
\textit{birthday} : \textit{NAME} \rightarrow\!\!\!\!\!\!+\ \textit{DATE} \\
\hline
\textit{known} = \operatorname{dom} \textit{birthday} \\
\hline
\end{array}
$$

[NAME; DATE] are basic types of the specification

Example: Birthday Book

- One possible state

$$\mathit{known} = \{\text{John}, \text{Mike}, \text{Susan}\}$$

$$\mathit{birthday} = \{\text{John} \mapsto \text{25–Mar}, \text{Mike} \mapsto \text{20–Dec}, \text{Susan} \mapsto \text{20–Dec}\}$$

BirthdayBook

$$\mathit{known} : \mathbb{P}\ \mathit{NAME}$$
$$\mathit{birthday} : \mathit{NAME} \rightarrowtail\!\!\!\!\!\!+ \ \mathit{DATE}$$

$$\mathit{known} = \mathrm{dom}\ \mathit{birthday}$$

- Stated properties
  - No limit on the number of birthdays recorded
  - No premature decision about the format of names and dates
  - Q: How many birthday can a person have?
  - Q: Does everyone have a birthday?
  - Q: Can two persons share the same birthday?

State Schema: More Examples

- Simple text editor with limited memory
- Editor state modeled by two state variables, the texts to the left and right of the cursor

[CHAR]

TEXT == seq CHAR

$$
\begin{array}{|l}
maxsize : \mathbb{N} \\
\hline
maxsize \leq 65535
\end{array}
$$

Editor

$$
\begin{array}{|l}
left, right : TEXT \\
\hline
\#\,(left \frown right) \leq maxsize
\end{array}
$$

How to Specify Dynamic Aspects?

- Use schemas to describe operations, *relationship* between their inputs (?) and outputs (!), *changes* of state that happen.

- Example: AddBirthday
  - Q: What're inputs, outputs, and the state components?
  - Q: What's the pre and post-conditions?
  - Q: What's the meaning of operations?

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

Δ And Ξ Notations

- Δ *BirthdayBook* state change of *BirthdayBook*
- Ξ *BirthdayBook* no state change of *BirthdayBook*

AddBirthday

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

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

BirthdayBook

$known : \mathbb{P}\ NAME$
$birthday : NAME \nrightarrow DATE$

$known = \mathrm{dom}\ birthday$

More Example: *FindBirthday*

- Use of Ξ notation
- Specify no state change

$$
\begin{array}{l}
\textit{FindBirthday} \\
\hline
\Xi \textit{BirthdayBook} \\
name? : NAME \\
date! : DATE \\
\hline
name? \in known \\
date! = birthday(name?)
\end{array}
$$

More Example: *Remind*

- Use of set comprehension notation
  - Selection (|) vs. collection (•)
- Q: What does it return?

$$
\begin{array}{l}
\textit{Remind} \\
\hline
\Xi \textit{BirthdayBook} \\
\textit{today}? : \textit{DATE} \\
\textit{cards}! : \mathbb{P}\ \textit{NAME} \\
\hline
\textit{cards}! = \{ n : \textit{known} \mid \textit{birthday}(n) = \textit{today}? \} \\
\hline
\end{array}
$$

More Example: *InitBirthdayBook*

- Describes the *initial state* of the system
- By convention, use Init as prefix

InitBirthdayBook

BirthdayBook

$$known = \varnothing$$

Stating and Proving Properties

- E.g., known' = known ∪ {name?}

$$
\begin{aligned}
& known' \\
& \quad = \operatorname{dom} birthday' && \text{(invariant after)} \\
& \quad = \operatorname{dom}(birthday \cup \{name? \mapsto date?\}) \\
& && \text{(specification of } AddBirthday\text{)} \\
& \quad = \operatorname{dom} birthday \cup \operatorname{dom} \{name? \mapsto date?\} \\
& && \text{(fact about `dom')} \\
& \quad = \operatorname{dom} birthday \cup \{name?\} && \text{(fact about `dom')} \\
& \quad = known \cup \{name?\} && \text{(invariant before)}
\end{aligned}
$$

BirthdayBook

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

$known = \operatorname{dom} birthday$

AddBirthday

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

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

Operators

- Λ (Conjunction of the two predicate parts) – any common variables of the two schemas are merged
- **V** (the effect of the schema operator is to make a schema in which the predicate part is the result of joining the predicate parts of its two arguments with the logical connective **V**).

Logical Conjunction Operator

The conjunction operator $\wedge$ of the schema calculus allows us to combine this description with our previous description of AddBirthday

AddBirthday $\wedge$ Success

This describes an operation which, for correct input, both acts as described by *AddBirthday* and produces the result *ok*.

More Examples

- Strengthening specifications by making partial operations total.
- Q: How to make *AddBirthday* total?

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

Strengthening *AddBirthday*

$REPORT ::= ok \mid already\_known$

$$
\begin{array}{|l}
\hline
\textbf{Success} \\
result! : REPORT \\
\hline
result! = ok \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textbf{AlreadyKnown} \\
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \in known \\
result! = already\_known \\
\hline
\end{array}
$$

$$
\begin{array}{l}
RAddBirthday \mathrel{\widehat=} \\
\quad (AddBirthday \land Success) \lor AlreadyKnown
\end{array}
$$

RAddBirthday

$RAddBirthday \mathrel{\widehat{=}}$
$\quad (AddBirthday \land Success) \lor AlreadyKnown$

RAddBirthday

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

$(name? \notin known \land$
$\quad birthday' = birthday \cup \{name? \mapsto date?\} \land$
$\quad result! = ok) \lor$
$(name? \in known \land$
$\quad birthday' = birthday \land$
$\quad result! = already\_known)$

Notice the framing constraint. Why?

Strengthening *FindBirthday* and *Remind*

$$
\begin{array}{|l}
\hline
\textit{FindBirthday} \\
\Xi \textit{BirthdayBook} \\
\textit{name?} : \textit{NAME} \\
\textit{date!} : \textit{DATE} \\
\hline
\textit{name?} \in \textit{known} \\
\textit{date!} = \textit{birthday}(\textit{name?}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Remind} \\
\Xi \textit{BirthdayBook} \\
\textit{today?} : \textit{DATE} \\
\textit{cards!} : \mathbb{P}\ \textit{NAME} \\
\hline
\textit{cards!} = \{\, n : \textit{known} \mid \textit{birthday}(n) = \textit{today?} \,\} \\
\hline
\end{array}
$$

RFindBirthday and *RRemind*

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

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{NotKnown}} \\
\hline
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \notin known \\
result! = not_known \\
\hline
\end{array}$$

$$RFindBirthday \mathrel{\widehat=} (FindBirthday \land Success) \lor NotKnown$$

$$RRemind \mathrel{\widehat=} Remind \land Success$$

Schema Calculus

- Modularize specifications by building large schemas from smaller ones, e.g.,
  - Separating normal operations from error handling
  - Separating access restrictions from functional behaviors
  - Promoting and framing operations, e.g., reading named a file from reading a file
  - …

  => Separation of concerns
- How?

  Provide operations for combining schemas, e.g.,

  $$S_1 \wedge S_2$$

  where S_1 and S_2 are schemas

Schema Calculus

- Schema operator for every logical connective and quantifier
- Conjunction and disjunction are most useful
- Merge declarations and combine predicates,

  $S_1 \mathrel{\widehat{=}} [D_1 \mid C_1]$

  $S_2 \mathrel{\widehat{=}} [D_2 \mid C_2]$

  $S_1 \wedge S_2 \equiv [D_1; D_2 \mid C_1 \wedge C_2]$

Example

Quotient

$n, d, q, r: \mathbb{N}$

$d \neq 0$
$n = q*d + r$

Remainder

$r, d: \mathbb{N}$

$r < d$

$Division \triangleq Quotient \wedge Remainder$

Division

$n,d,q,r: \mathbb{N}$

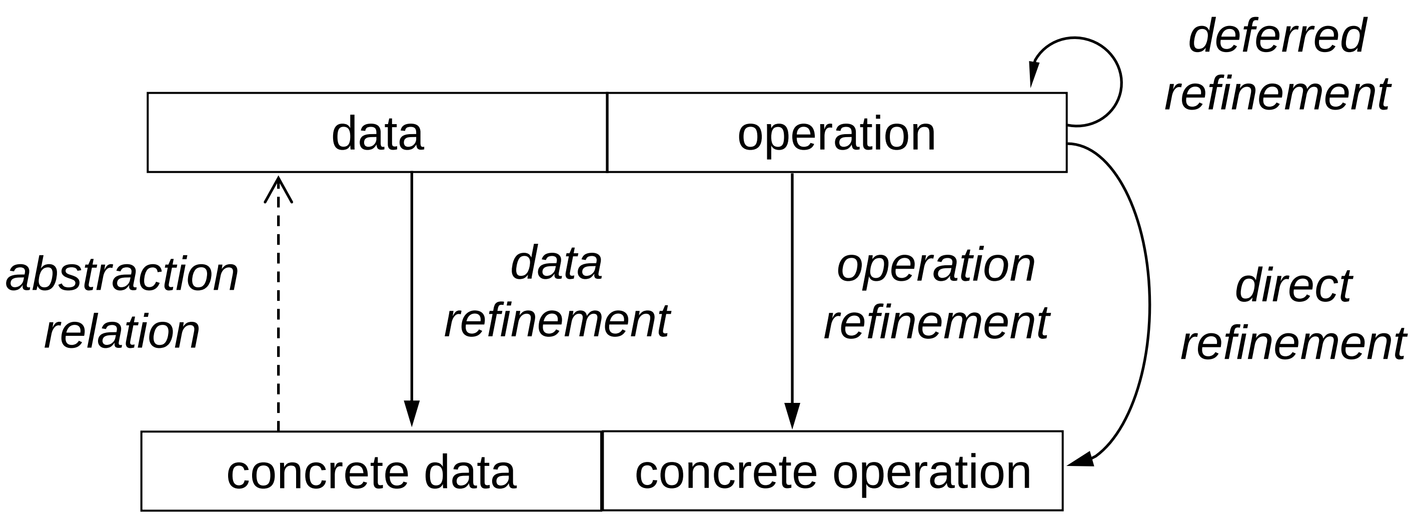
$d \neq 0$
$r < d$
$n = q*d + r$

Refinement---From Specification to Designs and Implementation

- Previously, Z to specify a software module
- Now, Z to document the design of a programs
- Key idea: *data refinement*
  - Describe concrete data structures (<-> abstract data in specification)
  - Derive descriptions of operations in terms of concrete data structures
  - Often data refinement leads to *operation refinement* or *algorithm development*

Specification Refinement

- Done in a single or multiple steps
- Referred to as direct refinement and deferred refinement



Implementation of Birthday Book

- Expressive clarity in abstract data structure
- Efficiency and representation in concrete data structure
- One possible representation

  NAME[] names;

  DATE[] dates;
- Q: Any better representation in Java?

BirthdayBook

$$known : \mathbb{P}\ NAME$$

$$birthday : NAME \pfun DATE$$

$$known = \mathrm{dom}\ birthday$$

Concrete State Model, *BirthdayBook1*

- Arrays modeled mathematically modeled as functions:

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

- I.e., *names*[*i*] as *names*(*i*) and *names*[*i*] = *v* as

$$names' = names \oplus \{i \mapsto v\}$$

$$
\begin{array}{|l}
\hline
\textit{BirthdayBook1} \\
names : \mathbb{N}_1 \rightarrow NAME \\
dates : \mathbb{N}_1 \rightarrow DATE \\
hwm : \mathbb{N} \\
\hline
\forall\, i, j : 1 \,..\, hwm \bullet \\
\quad i \neq j \Rightarrow names(i) \neq names(j) \\
\hline
\end{array}
$$

Abstraction Relation, *Abs*

- Relation between abstract state space and concrete state space, e.g., BirthdayBook and BirthdayBook1
- Q: Why abstract relation?

$$
\begin{array}{|l}
\hline
\textit{Abs} \\
\textit{BirthdayBook} \\
\textit{BirthdayBook1} \\
\hline
\textit{known} = \{ i : 1 \mathinner{..} \textit{hwm} \bullet \textit{names}(i) \} \\
\forall i : 1 \mathinner{..} \textit{hwm} \bullet \\
\quad \textit{birthday}(\textit{names}(i)) = \textit{dates}(i) \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\hline
\textit{BirthdayBook1} \\
\textit{names} : \mathbb{N}_1 \to \textit{NAME} \\
\textit{dates} : \mathbb{N}_1 \to \textit{DATE} \\
\textit{hwm} : \mathbb{N} \\
\hline
\forall i, j : 1 \mathinner{..} \textit{hwm} \bullet \\
\quad i \neq j \Rightarrow \textit{names}(i) \neq \textit{names}(j) \\
\hline
\end{array}
$$

Operation Refinement, *AddBirthday1*

- Manipulate *names* and *dates* arrays

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

$$
\begin{array}{|l}
\hline
\textit{BirthdayBook1} \\
\textit{names} : \mathbb{N}_1 \rightarrow \textit{NAME} \\
\textit{dates} : \mathbb{N}_1 \rightarrow \textit{DATE} \\
\textit{hwm} : \mathbb{N} \\
\hline
\forall\, i, j : 1 \mathinner{\ldotp\ldotp} \textit{hwm} \bullet \\
\quad i \neq j \Rightarrow \textit{names}(i) \neq \textit{names}(j) \\
\hline
\end{array}
$$

Correctness of Operation Refinement

- Whenever *AddBirthday* is legal in some abstract state, the implementation *AddBirthday1* is legal in any corresponding concrete state, i.e., $Pre_A \Rightarrow Pre_C$
- The final state which results from AddBirthday1 represents an abstract state which AddBirthday could produce, i.e., $Post_C \Rightarrow Post_A$

$$
\begin{tikzcd}
Pre_A \arrow[rr, "Op_A"] \arrow[d] & & Post_C \\
Pre_C \arrow[rr, "Op_C"] & & Post_C \arrow[u]
\end{tikzcd}
$$

Correctness of *AddBirthday1*

- $Pre_A \Rightarrow Pre_{C,}$ i.e., $name? \notin known$ $\quad \forall\, i : 1\,..\,hwm \bullet name? \neq names(i)$
- Does this hold? Yes, because:

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

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

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

Correctness of *AddBirthday1*

- $Post_C \Rightarrow Post_A$
- Read the proof (p. 46)

  $Abs(Post_C) \Rightarrow Post_A$

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

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

Implementation of *AddBirthday1*

```
void addBirthday(NAME name, DATE date) {
        hwm++;
        names[hwm] = name;
        dates[hwm] = date;
}
```

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

Refinement of *FindBirthday*

$$
\begin{array}{|l}
\hline
\textit{FindBirthday1} \\
\Xi \textit{BirthdayBook1} \\
name? : NAME \\
date! : DATE \\
\hline
\exists\, i : 1 \mathinner{\ldotp\ldotp} hwm \bullet \\
\quad name? = names(i) \land date! = dates(i) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{BirthdayBook1} \\
names : \mathbb{N}_1 \rightarrow NAME \\
dates : \mathbb{N}_1 \rightarrow DATE \\
hwm : \mathbb{N} \\
\hline
\forall\, i, j : 1 \mathinner{\ldotp\ldotp} hwm \bullet \\
\quad i \neq j \Rightarrow names(i) \neq names(j) \\
\hline
\end{array}
$$

Refinement of *Remind*

$$
\begin{array}{|l}
\hline
\textit{Remind1} \\
\Xi \textit{BirthdayBook1} \\
\textit{today?} : \textit{DATE} \\
\textit{cardlist!} : \mathbb{N}_1 \rightarrow \textit{NAME} \\
\textit{ncards!} : \mathbb{N} \\
\hline
\{ i : 1 \mathinner{\ldotp\ldotp} \textit{ncards!} \bullet \textit{cardlist!}(i) \} \\
\quad = \{ j : 1 \mathinner{\ldotp\ldotp} \textit{hwm} \mid \textit{dates}(j) = \textit{today?} \bullet \textit{names}(j) \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{AbsCards} \\
\textit{cards} : \mathbb{P}\ \textit{NAME} \\
\textit{cardlist} : \mathbb{N}_1 \rightarrow \textit{NAME} \\
\textit{ncards} : \mathbb{N} \\
\hline
\textit{cards} = \{ i : 1 \mathinner{\ldotp\ldotp} \textit{ncards} \bullet \textit{cardlist}(i) \} \\
\hline
\end{array}
$$

Refinement of *InitBirthdayBook*

$$
\begin{array}{|l}
\hline
\textit{InitBirthdayBook1} \\
\textit{BirthdayBook1} \\
\hline
hwm = 0 \\
\hline
\end{array}
$$

Race condition

We have not handled the condition when user tries to add a birthday, which is already known to the system, or tries to find the birthday of someone not known.

Handle this by adding an extra result! To each operation.

Result := of | already_known | not_known

Success

Result! : REPORT

Result! = ok

Logical Disjunction operator

AlreadyKnown

Ξ BirthdayBook

name? : NAME

result?: REPORT

Name? $\in$ known

Result! = already_known

This declaration specifies that if error occurs, the state of the system should not change.

Robust version of AddBirthday can be

RAddBirthday = (AddBirthday Λ Success) V Alreadyknown

Use of Operators

RAdd Birthday

ΔBirthday Book

name?: NAME

date?: DATE

result!: REPORT

(name? $\notin$ known $\wedge$

birthday'= birthday $\cup$ {name? $\longrightarrow$ Date?} $\wedge$

result!= ok) $\vee$

(name? $\in$ known $\wedge$

birthday' = birthday $\wedge$

result != already_known)

From specification to design

<u>Data Refinement</u>

" to describe the concrete data structures which the program will use to represent the abstract data in the specification, and to derive description of the operation in terms of the concrete data structures"

<u>Direct Refinement:</u> method to go directly from abstract specification to program in one step

Data Refinement

Data Structures:

Two arrays : names [1…] of NAME

dates [1…] of DATES

names' = names$\oplus${i $\longrightarrow$ v} ; names[i] := v the right side of this equation is a function which takes the same value as names everywhere except at the argument i, where it takes the value 'v'.

Example(Data and Direct Refinement)

FindBirthday1

Ξ BirthdayBook1
name?:NAME
date?:DATE

$\exists i : 1..\ hwm$
$\quad name? = names(i) \wedge date! = dates(i)$

```
Procedure FindBirthday(name: NAME; var date : DATE);
                    var i: INTEGER;
          begin
                    i:=1;
                    while names[i] ≠ name do i := i+1;
                    dates := dates[i]
          end;
```

Features Z Notation

- Is used to verify the specification
- Independent of program code
- Mathematical data model
- Allow to model a specification which can directly lead to the code.
- Represent both static and dynamic aspects of a system

Features Z Notation

- Decompose specification into small pieces (Schemas)
- Schemas are used to describe both static and dynamic aspects of a system
- Data Refinement
- Direct Refinement
- You can ignore details in order to focus on the aspects of the problem you are interested in
- ISO standard, ISO/IEC 13568:2002