An Introduction to the Z Notation

(Based on [J.Woodcock and J.Davies 1996; J.M. Spivey 1998])

Jing-Jie Lin

Dept. of Information Management
National Taiwan University

November 25, 2010

Agenda

What is Formal Specification

- Use mathematical notation to describe in a precise way the properties which an information system must have, without unduly constraining the way in which these properties are achieved.
- Formal specifications describe *what* the system must do without saying *how* it is to be done.
- A formal specification can serve as a single, reliable reference point for those
    - who investigate the customer's needs,
    - who implement programs to satisfy those needs,
    - who test the results, and
    - who write instruction manuals for the system.

Specification Qualities

A good specification should be

- abstract and complete.
- clear and unambiguous.
- concise and comprehensible.
- easy to maintain and cost-effective.

Agenda

What is Z Notation

- Z(Zed) is a formal specification language used for describing and modeling computing systems.
- The Z notation is based on
  - The *mathematical language* is used to describe objects and their properties. (e.g., sets, logic, and relations)
  - Mathematical objects and their properties can be collected together in schema. The *schema language* is used to describe the state of a system, and the ways in which that state may change.
  - The *theory of refinement*: the mathematical data types of specification to be implemented by more computer-oriented data type in a design.

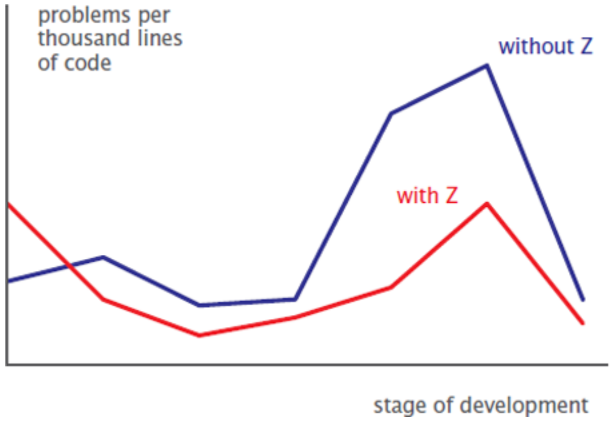
What is Z Notation

We can use Z to

- describe data structures.
- model system state.
- explain design intentions.
- verify development steps.

What is Z Notation

Qualitative Results



Mathematical Language

- Sets
- Relations
- Functions
- Numbers and finiteness

Mathematical Language: Sets

- *Set comprehension*:
  Given any non-empty set s, we can define a new set by considering only those elements of s that satisfy some property p.
- Denote the set of elements x in s that satisfy predicate p.

$$\{x : s \mid p\}$$

- Example: suppose that a red car is seen driving away from the scene of a crime. If *Person* denotes the set of all people, then the set to consider is given by

$$\{x : \mathit{Person} \mid x \text{ drives a red car}\}$$

Mathematical Language: Sets

- *Term comprehension*:
  We may also describe a set of objects constructed from certain elements of a given set.
- Denote the set of all expressions e such that x is drawn from s and satisfies p.

$$\{x : s \mid p \bullet e\}$$

- Example: In order to pursue their investigation of the crime, the authorities require a set of addresses to visit. This set is given by

$$\{x : \mathit{Person} \mid x \text{ drives a red car} \bullet \mathit{address}(x)\}$$

Mathematical Language: Sets

- A comprehension without a term part is equivalent to one in which the term is the same as the bound variable:

$$\{x : s \mid p\} == \{x : s \mid p \bullet x\}$$

- The comprehension without a predicate part is equivalent to the one with the predicate true:

$$\{x : s \bullet e\} == \{x : s \mid true \bullet e\}$$

Mathematical Language: Sets

- Denote the set of expression e formed as x and y range over a and b, respectively, and satisfy predicate p.

  $$\{x : a;\ y : b \mid p \bullet e\}$$

- Example: an eyewitness account has established that the driver of the red car had an accomplice, and that this accomplice left a copy of the Daily Mail at the scene:

  $$\begin{aligned}&\{x : \mathit{Person};\ y : \mathit{Person} \mid x \text{ is associated with } y\\ &\wedge\ x \text{ drives a red car}\\ &\wedge\ y \text{ reads the Daily Mail} \bullet x\}\end{aligned}$$

Mathematical Language: Sets

- *Power set*:
  If *a* is a set, then the set of all subsets of *a* is called the *power set* of *a*, and written $\mathbb{P}\ a$.
- Example:
  - $\mathbb{P}\ \{x,y\} = \{\ \emptyset,\{x\},\{y\},\{x,y\}\}$
  - $\{1,2,3,4\} \in \mathbb{P}\ \mathbb{N}$

Mathematical Language: Sets

- *Cartesian product :*
  If X and Y are sets, then the Cartesian product X × Y is the set of all possible ordered pairs (x,y), where x is an element of X and y is an element of Y:

  $$X \times Y = \{(x, y) \mid x \in X \textit{ and } y \in Y\}$$

- Example:
  - $\{1,2\} \times \{3,4\} = \{(1,3),(1,4),(2,3),(2,4)\}$

Mathematical Language: Sets

- *Types* :
  A type is a maximal set, at least within the confines of the current specification.
  The Z notation has a single built-in type: the set of all integers $\mathbb{Z}$:
  $$\mathbb{Z} = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$$
- Any other types may be constructed from $\mathbb{Z}$, or from user-defined basic types.
- Every expression that appears in Z specification is associated with a unique type, and if the expression is defined, then the value of the expression is a member of its type.

Mathematical Language: Relations

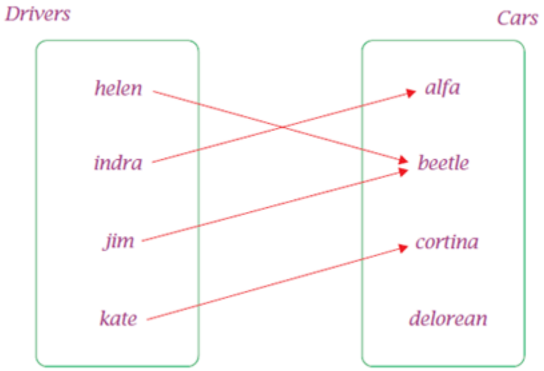
- Binary relations
  Denotes the set of all relations between X and Y:

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

- Maplet
  The pair (x,y) can be written as x $\mapsto$ y.

$$\begin{array}{|l}
[X, Y] \\
\_ \mapsto \_ : X \times Y \rightarrow X \times Y \\
\hline
\forall x : X;\ y : Y \bullet x \mapsto y = (x, y)
\end{array}$$

Mathematical Language: Relations



$$\_drives\_ : Drivers \leftrightarrow Cars$$

$$drives = \{helen \mapsto beetle, indra \mapsto alfa, jim \mapsto beetle, kate \mapsto cortina\}$$

Mathematical Language: Relations

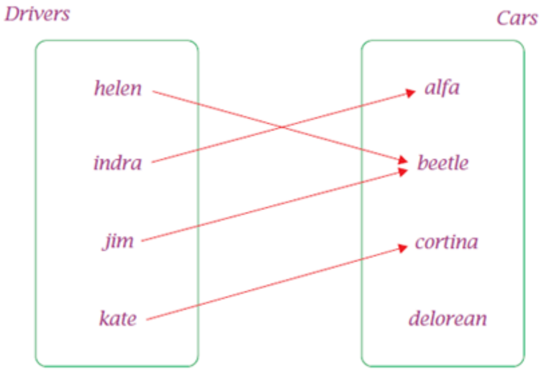
- Domain and Range

$$\operatorname{dom} R = \{x : X;\ y : Y \mid x \mapsto y \in R \bullet x\}$$

$$\operatorname{ran} R = \{x : X;\ y : Y \mid x \mapsto y \in R \bullet y\}$$

Mathematical Language: Relations

- Domain and Range Example: Function-Drives



$$dom\ drives = \{helen, indra, jim, kate\}$$

$$ran\ drives = \{alfa, beetle, cortina\}$$

Mathematical Language: Relations

- Domain Subtraction

$$A \mathbin{⩤} R = \{x : X;\ y : Y \mid x \mapsto y \in R \land x \notin A \bullet x \mapsto y\}$$

- An example of domain subtraction
If we are concerned only with people who are not called 'Helen', then the relation $\{Henlen\} \mathbin{⩤} Drives$ tells us all that we want to know. It is a relation with three elements:

$$\{Indra \mapsto alfa, Jim \mapsto beetle, Kate \mapsto cortina\}$$

Mathematical Language: Relations

- Domain $\operatorname{dom} R = \{x : X;\ y : Y \mid x \mapsto y \in R \bullet x\}$
- Range $\operatorname{ran} R = \{x : X;\ y : Y \mid x \mapsto y \in R \bullet y\}$
- Domain Restriction
  $A \lhd R = \{x : X;\ y : Y \mid x \mapsto y \in R \land x \in A \bullet x \mapsto y\}$
- Range Restriction
  $R \rhd B = \{x : X;\ y : Y \mid x \mapsto y \in R \land y \in B \bullet x \mapsto y\}$
- Domain Subtraction
  $A \ntriangleleft R = \{x : X;\ y : Y \mid x \mapsto y \in R \land x \notin A \bullet x \mapsto y\}$
- Range Subtraction
  $R \ntriangleright B = \{x : X;\ y : Y \mid x \mapsto y \in R \land y \notin B \bullet x \mapsto y\}$

Mathematical Language: Functions

- Partial functions
  From X to Y is a relation that maps each element of X to at most one element of Y. The element of Y, if it exists, is written *f(x)*.

$$X \pfun Y == \{f : X \leftrightarrow Y \mid \forall x : X;\ y_1, y_2 : Y \bullet (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2\}$$

- Total functions
  The set of total functions are partial functions whose domain is the whole of X. They relate each element of X to exactly one element of Y.

$$X \rightarrow Y == \{f : X \pfun Y \mid \mathrm{dom}\, f = X\}$$

Mathematical Language: Functions

- Partial Functions: each element of the source set is mapped to at most one element of the target.
  Total Functions: each element of the source set is mapped to some element of the target.
- Injective (1 to 1): each element of the domain is mapped to a different element of the target.
  - $\rightarrowtail\!\!\!\!\!\!+$: partial, injective functions
  - $\rightarrowtail$: total, injective functions
- Surjective (onto): the range of the function is the whole of the target
  - $+\!\!\!\!\!\twoheadrightarrow$: partial, surjective functions
  - $\twoheadrightarrow$: total, surjective functions
- Bijective (1 to 1 correspondence): both injective and surjective
  - $\rightarrowtail\!\!\!\!\!\twoheadrightarrow$: total, bijective functions

Mathematical Language: Functions

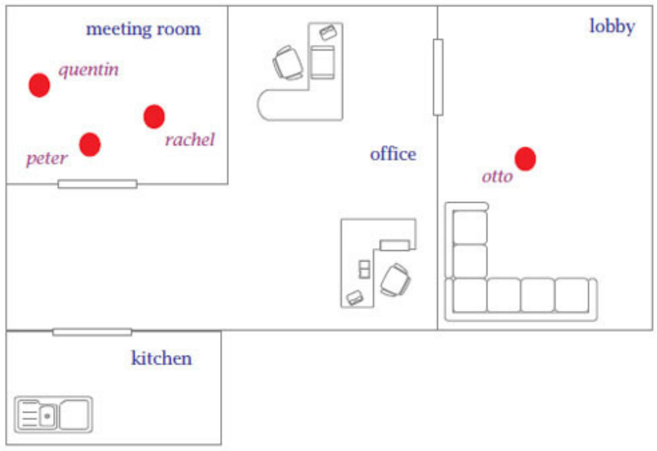
- Overriding
  If f and g are functions of the same type, then $f \oplus g$ is a function that agrees with f everywhere outside the domain of g; but agrees with g where g is defined.

$$[X, Y]$$
$$\_ \oplus \_ : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$$
$$\forall f, g : X \leftrightarrow Y \bullet$$
$$f \oplus g = (\mathrm{dom}\, g \lhd f) \cup g$$

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

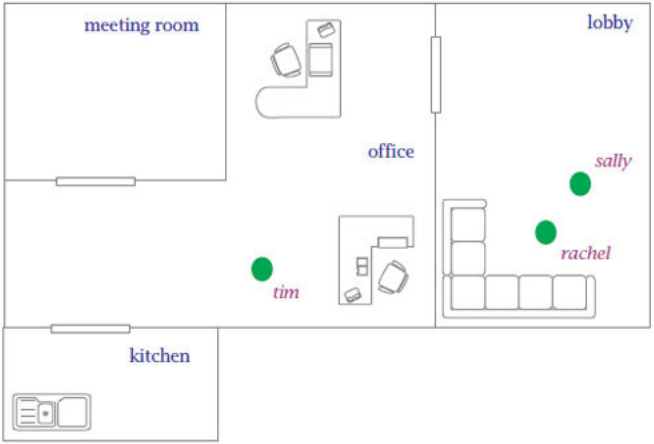
Overriding

Original



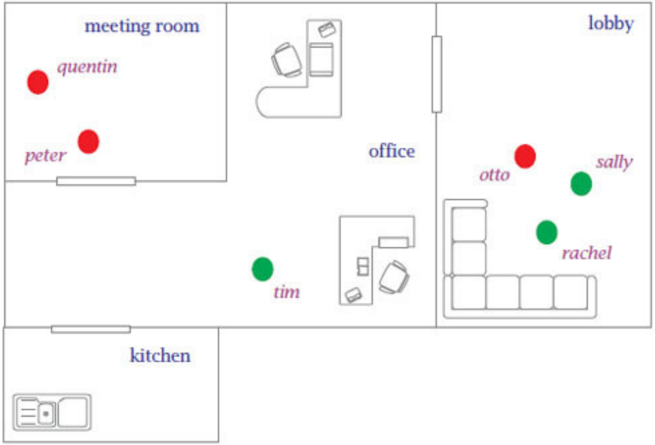
Overriding

Update



Overriding

Override



Mathematical Language: Numbers and finiteness

- Natural numbers

$$\mathbb{N} == \{n : \mathbb{Z} \mid n \geq 0\}$$

- Strictly positive integers

$$\mathbb{N}_1 == \mathbb{N} \setminus \{0\}$$

Agenda

Schema Language

We can write the text of a schema in one of two the following two forms:

```
┌─ Name ──────────────────────────────
│ declaration
├──────────────
│ constraint
└─────────────────────────────────────
```

or

$$Name \mathrel{\widehat{=}} [declaration \mid constraint]$$

Schema Language

$$\mathit{Name} \mathrel{\widehat{=}} [a : \mathbb{Z};\ c : \mathbb{P}\,\mathbb{Z} \mid c \neq \phi \land\ a \in c]$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Name}} \\ \hline
a : \mathbb{Z} \\
c : \mathbb{P}\,\mathbb{Z} \\ \hline
c \neq \phi \\
a \in c \\ \hline
\end{array}
$$

Schema Language

We can use the language of schemas to describe the state of a system, and operation upon it.
Suppose that the state of a system is modeled by the following schema

```
┌─ State ──────────────────────────────
│ a : A
│ b : B
├──────────
│ P
└──────────────────────────────────────
```

Schema Language

To describe an operation upon the state, we use two copies of *State*: one representing the state before the operation; the other representing the state afterwards.

$$\begin{array}{|l}
\hline
\textit{State}' \\
a' : A \\
b' : B \\
\hline
P[a'/a, b'/b] \\
\hline
\end{array}$$

The constraint part of the schema is modified to reflect the new names of the state variables.

Schema Language

Then we can describe an operation by including both *State* and *State*' in the declaration part of a schema. For example,

```
┌─ Operation ─────────────────────────────
│ State
│ State′
│ i? : I
│ o! : O
├──────────────
│ . . .
└─────────────────────────────────────────
```

The behavior of the operation is described in the constraint part of the schema.
Note that the schema also includes an input component of type I and an output component of type O.

Schema Language

When a schema name appears in a declaration part of a schema, the result is a merging of declarations and a conjunction of constraints.

```
┌─ OperationOne ───────────────────
│ State
│ State′
└──────────────────────────────────
```

$$\begin{array}{|l}\hline \textit{OperationOne} \\ \textit{State} \\ \textit{State}' \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{OperationTwo} \\ a, a' : A \\ b, b' : B \\ \hline P \\ P[a'/a, b'/b] \\ \hline\end{array}$$

Schema Language

Δ *Schema* can be applied whenever we wish to describe an operation that may change the state.

$$\begin{array}{|l}
\hline
\Delta\, Schema \\
Schema \\
Schema' \\
\hline
\end{array}$$

Ξ *Schema* can be applied whenever we wish to describe an operation that does not change the state.

$$\begin{array}{|l}
\hline
\Xi\, Schema \\
\Delta\, Schema \\
\hline
\theta\, Schema = \theta\, Schema' \\
\hline
\end{array}$$

Note: θ here means the valuation of variables in the schema.

Schema Language

- Different aspects of the state can be described as separate schemas; these schemas may be combined in various ways using *schema operators*:
  - The logical schema operators:

    $$\begin{array}{l} \wedge \\ \vee \\ \neg \\ \forall \\ \exists \end{array}$$

  - The relational schema operators:

    $\fatsemi$ $-$*Sequential composition*
    $\gg$ $-$ *Piping*

Schema Language

- If S and T are two schemas, then their conjunction $S \wedge T$ is a schema
  - whose declaration is a merge of the two declarations.
  - whose constraint is a conjunction of the two constraints.
- Their disjunction $S \vee T$ is a schema
  - whose declaration is a merge of the two declarations.
  - whose constraint is a disjunction of the two constraints.

Schema Language

S

$a : A$

$b : B$

P

T

$b : B$

$c : C$

Q

Schema Language

The schema $S \land T$ (conjunction) is equivalent to

$$
\begin{array}{|l}
\hline
\quad S \land T \\
\quad a : A \\
\quad b : B \\
\quad c : C \\
\hline
\quad P \land Q \\
\hline
\end{array}
$$

The schema $S \lor T$ (disjunction) is equivalent to

$$
\begin{array}{|l}
\hline
\quad S \lor T \\
\quad a : A \\
\quad b : B \\
\quad c : C \\
\hline
\quad P \lor Q \\
\hline
\end{array}
$$

Agenda

The Birthday Book

Basic three functions:

- Add new birthday-name record.
- Find the birthday of a person.
- Give a date, return names of people whose birthday is exactly that day.

The Birthday Book

Given basic types:

$$[NAME, DATE]$$

Use a schema to describe the state of the birthday book:

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

- *known* is the set of names with birthdays recorded.
- *birthday* is a function when applied to certain names, gives the birthdays associated with them.
- *invariant* is relationship which is true in every state of the system.

The Birthday Book

One possible state of the system has three people in the set *known*, with their birthdays recorded by the function *birthday*:

$$\begin{aligned}&known = \{Cindy, Randy, John\}\\&birthday =\\&\{Cindy \mapsto 7/5,\\&\ Randy \mapsto 11/5,\\&\ John \ \mapsto 6/2\}.\end{aligned}$$

The invariant is satisfied, because *birthday* records a date for exactly the three names in *known*.

The Birthday Book

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

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

The Birthday Book

Specify an operation to add new birthday-name record:

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

The Birthday Book

We can prove $\mathit{known}' = \mathit{known} \cup \{\mathit{name}?\}$ from the specification of *AddBirthday*, using the invariants on the state before and after the operation:

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

Note: Laws of Domain
$\operatorname{dom}\{Q \cup R\} = \operatorname{dom}\{Q\} \cup \operatorname{dom}\{R\}$
$\operatorname{dom}\{x_1 \mapsto y_1, .., x_1 \mapsto x_n\} = \{x_1, .., x_n\}$

The Birthday Book

Find the birthday of a person:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FindBirthday}} \\
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{date}! : \textit{DATE} \\
\hline
\textit{name}? \in \textit{known} \\
\textit{date}! = \textit{birthday}(\textit{name}?) \\
\hline
\end{array}
$$

The Birthday Book

Give a date, return names of people whose birthday is exactly that day.

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

The Birthday Book

To finish the specification, we must say what state the system is in when it is first started. This is the initial state of the system, and it also is specified by a schema:

```
┌─ InitBirthdayBook ─────────────────────────
│ BirthdayBook
├──────────────
│ known = ∅
└────────────────────────────────────────────
```

Agenda

- What Is Formal Specification
- What Is Z Notation
  - Mathematical Language
  - Schema Language
- Example: the Birthday Book
- Strengthening the Specification
- Implementing the Birthday Book

Strengthening the Specification

- A correct implementation of our specification will faithfully record birthdays and display them, so long as there are no mistakes in the input. But the specification has a serious flaw:
  - add a birthday for someone already known to the system.
  - find the birthday of someone not known.
- The specification we have described clearly and concisely the behavior for correct input, and modifying it to describe the handling of incorrect input could only make it obscure.

Strengthening the Specification

- Better solution :
  - describe, separately from the first specification, the errors which might be detected and the desired responses to them.
  - use *schema operators* (e.g., $\wedge$, $\vee$) to combine the two descriptions into a stronger specification.
- Add an extra output *result*! to each operation on the system. When an operation is successful, this output will take the value *ok*, but it may take other values when an error is detected. The following free type definition defines *REPORT* to be a set containing exactly these three values:

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

Free Type

- Free type adds nothing to the power of Z, but it makes it easier to describe recursive structures such as lists and trees.
- A *free type* T is defined as follows:

$$T ::= c_1 \mid ... \mid c_m \mid d_1 \langle\!\langle E_1 \rangle\!\rangle \mid ... \mid d_n \langle\!\langle E_n \rangle\!\rangle$$

where disjoint $\langle \{c_1\}, ..., \{c_m\}, \mathrm{ran}\, d_1, ..., \mathrm{ran}\, d_n \rangle$,
c_1, ..., c_m are constant expressions,
d_1, ..., d_m are constructor functions, and
E_1, ..., E_m are expressions that may depend on set T.

Free Type Example

- Example:
  - The following *free type* definition, with seven distinct constants, is a structure of colors of the rainbow:

    $$Colors ::= red \mid orange \mid yellow \mid green \mid blue \mid indigo \mid violet$$

  - The following *free type* definition introduces a new type constructed using a single constant zero and a single constructor function succ:

    $$nat ::= zero \mid succ\langle\!\langle nat \rangle\!\rangle$$

  - This type has a structure which is exactly that of the natural numbers (zero corresponds to 0, and succ corresponds to the function +1).

Strengthening the Specification

We can define a schema Success which just specifies that the result should be *ok*:

```
__ Success ____________________________
| result! : REPORT
|______________
| result! = ok
|______________________________________
```

Then we can combine *AddBirthday* operation with *Success* by conjunction operator $\wedge$:

$$AddBirthday \wedge Success$$

This describes an operation for correct input.

Strengthening the Specification

Here is an operation which produces the report *already_known* when its input $name?$ is already a member of *known*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AlreadyKnown}} \\
\hline
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{result}! : \textit{REPORT} \\
\hline
\textit{name}? \in \textit{known} \\
\textit{result}! = \textit{already\_known} \\
\hline
\end{array}
$$

We can combine this description with the previous one to give a specification for a robust version of *AddBirthday*:

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

Strengthening the Specification

$$
\begin{array}{|l}
\textit{RAddBirthday} \\
\hline
\Delta \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{date}? : \textit{DATE} \\
\textit{result}! : \textit{REPORT} \\
\hline
(\textit{name}? \notin \textit{known} \land \\
\qquad \textit{birthday}' = \textit{birthday} \cup \{\textit{name}? \mapsto \textit{date}?\} \land \\
\qquad \textit{result}! = \textit{ok}) \lor \\
(\textit{name}? \in \textit{known} \land \\
\qquad \textit{birthday}' = \textit{birthday} \land \\
\qquad \textit{result}! = \textit{already\_known}) \\
\hline
\end{array}
$$

Strengthening the Specification

A robust version of the *FindBirthday* operation must be able to report if the input name is not known:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NotKnown}} \\
\hline
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{result}! : \textit{REPORT} \\
\hline
\textit{name}? \notin \textit{known} \\
\textit{result}! = \textit{not\_known} \\
\hline
\end{array}
$$

The robust operation either behaves as described by *FindBirthday* and reports success, or reports that the name was not known:

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

Strengthening the Specification

The *Remind* operation never results in an error, so the robust version need only add the report of success.

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

Agenda

Implementing the Birthday Book

- When a program is developed from a specification, two sorts of design decision usually need to be taken:
  - The data described by mathematical data types in the specification must be implemented by data structures of the programming language
  - The operations described by predicates in the specification must be implemented by algorithms expressed in a programming language
- Refinement:
  - *Data refinement* relates an *abstraction data type* (e.g., sets) to a *concrete data type* (e.g., arrays).
  - *Operation refinement* converts a specification of an operation on a system into an implementable program (e.g., a procedure).

Implementing the Birthday Book

- We choose to represent the birthday book with two arrays, which might be declared by:

  names: array [1..] of *NAME*
  dates: array [1..] of *DATE*

- These arrays can be modeled mathematically by functions from the set $\mathbb{N}_1$ of strictly positive integers to *NAME* or *DATE*:

$$\begin{aligned} names &: \mathbb{N}_1 \rightarrow NAME \\ dates &: \mathbb{N}_1 \rightarrow DATE \end{aligned}$$

Implementing the Birthday Book

The element $names[i]$ of the array is simply the value $names(i)$ of the function, and the assignment $names[i] := v$ is exactly described by the specification:

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

Implementing the Birthday Book

We describe the state space of the program as a schema. There is another variable *hwm* (for 'high water mark'); it shows how much of the arrays is in use.

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

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

Implementing the Birthday Book

We can document this with a schema Abs (abstraction schema) that defines the *abstraction relation* between the abstract state space *BirthdayBook* and the concrete state space *BirthdayBook*1:

$$
\begin{array}{|l}
\textbf{Abs} \\
\hline
\mathit{BirthdayBook} \\
\mathit{BirthdayBook}1 \\
\hline
\mathit{known} = \{i : 1..\mathit{hwm} \bullet \mathit{names}(i)\} \\
\forall\, i : 1..\mathit{hwm} \bullet \mathit{birthday}(\mathit{names}(i)) = \mathit{dates}(i) \\
\hline
\end{array}
$$

Implementing the Birthday Book

To add a new name, we increase *hwm* by one, and fill in the name and date in the arrays:

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

Note: Relationships of *AddBirthday*
$\textit{name}? \notin \textit{known}$

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

Correct Implementation

- Suppose *Aop* is a schema describing a specification and *Cop* is a schema describing the action of a program. *Abs* relates abstract and concrete states.
- A concrete schema is a correct implementation of abstract schema when
  - $pre\ Aop \wedge Abs \Rightarrow pre\ Cop$
    (ensures that the concrete operation terminates whenever the abstract operation is guaranteed to terminate)
  - $pre\ Aop \wedge Abs \wedge Cop \Rightarrow Abs' \wedge Aop$
    (ensures that the state after the concrete operation represents one of those abstract states in which the abstract operation could terminate)
- In this situation we shall write $Spec \sqsubseteq Ref$
  (The sign '$\sqsubseteq$' is the sign of refinement relation.)

Implementing the Birthday Book

- To show that *AddBirthday*1 is a correct implementation of *AddBirthday*, we have the following two proof obligations.
  - $pre\ AddBirthday \land Abs \Rightarrow pre\ AddBirthday1$
  - $pre\ AddBirthday \land Abs \land AddBirthday1 \Rightarrow Abs' \land AddBirthday$

The First Statement

- The pre $\mathit{AddBirthday}$ is $\mathit{name}? \notin \mathit{known}$.
  The pre $\mathit{AddBirthday}1$ is $\forall\, i : 1..\mathit{hwm} \bullet \mathit{names}? \neq \mathit{names}(i)$.
  Abs tells us that $\mathit{known} = \{i : 1..\mathit{hwm} \bullet \mathit{names}(i)\}$.
- This given
  $\mathit{name}? \notin \mathit{known} \wedge \mathit{known} = \{i : 1..\mathit{hwm} \bullet \mathit{names}(i)\}$
  $\Rightarrow \forall\, i : 1..\mathit{hwm} \bullet \mathit{names}? \neq \mathit{names}(i)$
- So the first proof obligation
  $\mathit{pre}\ \mathit{AddBirthday} \wedge \mathit{Abs} \Rightarrow \mathit{pre}\ \mathit{AddBirthday}1$ is true.

The Second Statement

- Think about the concrete states before and after an execution of *AddBirthday*1, and the abstract states they represent according to *Abs*.
- The two concrete states are related by *AddBirthday*1, and we must show that the two abstract states are related as prescribed by *AddBirthday*:

$$\textit{Prove that } birthday' = birthday \cup \{name? \mapsto date?\}$$

The Second Statement (Cont'd)

- The domains of these two functions are the same, because

$$
\begin{aligned}
&\operatorname{dom} birthday' \\
&\quad = known' && \text{[invariant after]} \\
&\quad = \{i : 1..hwm' \bullet names'(i)\} && \text{[from } Abs'\text{]} \\
&\quad = \{i : 1..hwm \bullet names'(i)\} \cup \{names'(hwm')\} \\
& && [hwm' = hwm + 1] \\
&\quad = \{i : 1..hwm \bullet names(i)\} \cup \{names?\} \\
& && [names' = names \oplus \{hwm' \mapsto names?\}] \\
&\quad = known \cup \{names?\} && \text{[from } Abs\text{]} \\
&\quad = \operatorname{dom} birthday \cup \{names?\} && \text{[invariant before]}
\end{aligned}
$$

Note: Laws of Domain
$\operatorname{dom}\{x_1 \mapsto y_1, .., x_1 \mapsto x_n\} = \{x_1, .., x_n\}$

The Second Statement (Cont'd)

- There is no change in the part of arrays which was in use before the operation.
  So for all i in the range 1..hwm:
  $$names'(i) = names(i) \land dates'(i) = dates(i)$$
- For any i in this range,

$$\begin{aligned} & birthday'(names'(i)) \\ & \quad = dates'(i) && \text{[from } Abs'\text{]} \\ & \quad = dates(i) && \text{[dates unchanged]} \\ & \quad = birthday(names(i)) && \text{[from } Abs\text{]} \end{aligned}$$

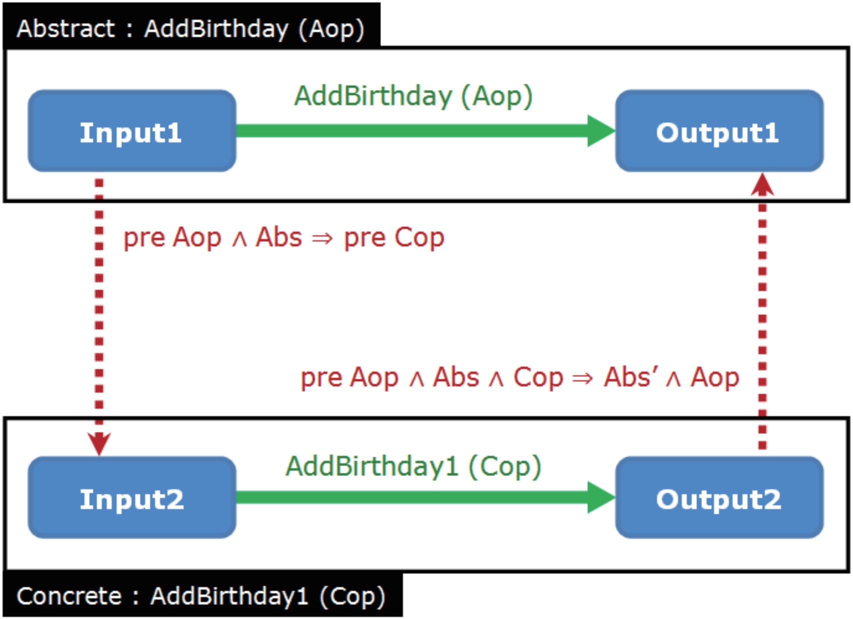
The Second Statement (Cont'd)

- For the new name, stored at index $hwm' = hwm + 1$

$$\begin{aligned} & birthday'(names?) \\ & \quad = birthday'(names'(hwm')) \quad [names'(hwm') = name?] \\ & \quad = dates'(hwm') \quad [\text{from } Abs'] \\ & \quad = date? \quad [\text{spec. of } Addbirthday1] \end{aligned}$$

- The second proof obligation $pre\ AddBirthday \land Abs \land AddBirthday1 \Rightarrow Abs' \land AddBirthday$ is also true.
- It shows that both of the proof obligation is true, so we can conclude that $AddBirthday1$ is a correct implementation of $AddBirthday$.

Refinement of the Birthday Book


Abstract : AddBirthday (Aop)
Input1
AddBirthday (Aop)
Output1
pre Aop ∧ Abs ⇒ pre Cop
pre Aop ∧ Abs ∧ Cop ⇒ Abs′ ∧ Aop
Input2
AddBirthday1 (Cop)
Output2
Concrete : AddBirthday1 (Cop)

Implementing the Birthday Book

The second operation, *FindBirthday*, is implemented by the following operation, again described in terms of the concrete state:

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

Check the pre-conditions and output

$$\begin{aligned}
&\textit{date}! = \textit{dates}(i) && [\text{spec. of } \textit{FindBirthday1}] \\
&= \textit{birthday}(\textit{names}(i)) && [\text{from } \textit{Abs}] \\
&= \textit{birthday}(\textit{name}?) && [\text{spec. of } \textit{FindBirthday1}]
\end{aligned}$$

Note: Relationships of *FindBirthday*
$\textit{name}? \in \textit{known}$
$\textit{date}! = \textit{birthday}(\textit{name}?)$

Implementing the Birthday Book

The operation *Remind* poses a new problem, because its output cards is a set of names. Here is a schema *AbsCards* that defines the abstraction relation:

AbsCards

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

Implementing the Birthday Book

The concrete operation can now be described: it produces as outputs *cardlist* and *ncards*:

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

Note: Relationships of *Remind*
$\textit{names!} = \{ n : \textit{known} \mid \textit{birthday}(n) = \textit{today?} \}$

Implementing the Birthday Book

The initial state of the program has $hwm = 0$:

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

$$
\begin{aligned}
& known \\
& \quad = \{ i : 1..hwm \bullet names(i) \} && [\text{from } Abs] \\
& \quad = \{ i : 1..0 \bullet names(i) \} && [\text{from } InitBirthdayBook1] \\
& \quad = \emptyset && [1..0 = \emptyset]
\end{aligned}
$$

Note: Relationships of *InitBirthdayBook*
$known = \emptyset$

Thank you for listening