CHAPTER 4
Z Schema Operators & Error Scenarios

Chapter Outline

- Z Schema Operators
  - Conjunction
  - Disjunction
  - Negation
  - Other schema operators

- Error Scenarios
- Complete Schema

Z Schema Operators

Schema Conjunction (^)

- If *S* and *T* are two schemas then their ***conjunction***, **S ∧ T** is also a schema
- The declaration is a merging of the **two** declaration parts and conjoining their predicate parts.
- The result is a schema that introduces both sets of variables and imposes both constraints.
- Allows us to specify different aspects of a specification separately, and then combine them to form a complete description.

Schema Conjunction (^)

- Example:

S
a: A
b: B

P

T
b: B
c: C

Q

$S \wedge T$ is equivalent to:

a: A
b: B
c: C

$P \wedge Q$

- The types for **b** must match, if not $S \wedge T$ will be undefined.

Schema Conjunction (^)

- Consider the schemas:

Quotient

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

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

Remainder

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

$$r < d$$

Schema Conjunction (^)

- To form a schema for Division:

Division ≙ Quotient ^ Remainder

$$
\begin{array}{|l}
\hline
\textit{Division} \\
n, d, q, r : \mathbb{N} \\
\hline
d \neq 0 \\
r < d \\
n = q * d + r \\
\hline
\end{array}
$$

Schema Disjunction ($\vee$)

- If *S* and *T* are two schemas then their ***disjunction***, **S $\vee$ T** is also a schema
- In which the declaration is a merging of the **two** declaration parts but the predicate parts are disjoined.
- Allows us to describe alternatives in the behaviour of a system.

Schema Disjunction (∨)

- Example:

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

```
 ┌─ T ─────────────
 │ b: B
 │ c: C
 ├──────────────
 │ Q
 └─────────────────
```

$S \vee T$ is equivalent to:

```
 ┌────────────────
 │ a: A
 │ b: B
 │ c: C
 ├────────────────
 │ P ∨ Q
 └─────────────────
```

The types for **b** must match, if not $S \vee T$ will be undefined.

Schema Disjunction (˅)

- Example:

  Schema for division by zero

$$
\begin{array}{|l}
\textit{DivideByZero} \\
\hline
d, q, r : \mathbb{N} \\
\hline
d = 0 \land q = 0 \land r = 0 \\
\hline
\end{array}
$$

- The total operation for division is given by:

  T_Division ≙ Division ˅ DivideByZero

$$
\begin{array}{|l}
\textit{T\_Division} \\
\hline
n, d, q, r : \mathbb{N} \\
\hline
(d \neq 0 \land r < d \land n = q * d + r) \lor \\
(d = 0 \land q = 0 \land r = 0) \\
\hline
\end{array}
$$

Schema Negation (¬)

- If *S* is a schema then its ***negation***, **¬S** is also a schema
- In which the declaration the same as that of *S*, **¬S** may be obtained by negating the predicate part.

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

¬ S is equivalent to:

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

Schema Inclusion

- Reuse the name of one schema in the **declaration part** of another schema
- When a schema name appears in a declaration part of a schema, the result is a merging of declarations and a conjunction of constraints.

Schema Inclusion

- Assume we have two schemas:

$$
\begin{array}{|l}
\hline
\textit{BoxOffice} \\
\hline
sold : SEAT \rightarrowtail\!\!\!\!\!\!+ \; CUSTOMER \\
seating : \mathbb{P}\; SEAT \\
\hline
\mathrm{dom}\; sold = seating \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Friends} \\
\hline
friends : \mathbb{P}\; CUSTOMER \\
status : STATUS \\
sold : SEAT \rightarrowtail\!\!\!\!\!\!+ \; CUSTOMER \\
\hline
status = premiere \Rightarrow \mathrm{ran}\; sold \subseteq friends \\
\hline
\end{array}
$$

Schema Inclusion

- EnhancedBoxOffice could have been defined as:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{EnhancedBoxOffice}} \\
\hline
\mathit{BoxOffice} \\
\mathit{status} : \mathit{STATUS} \\
\mathit{friends} : \mathbb{P}\ \mathit{CUSTOMER} \\
\hline
\mathit{status} = \mathit{premiere} \Rightarrow \mathrm{ran}\ \mathit{sold} \subseteq \mathit{friends} \\
\hline
\end{array}
$$

BoxOffice indicates that the schema is included into EnhancedBoxOffice schema

Schema Inclusion

- Or even as:

$$\begin{array}{|l}
\hline \textit{EnhancedBoxOffice} \\
\textit{BoxOffice} \\
\textit{Friends} \\
\hline
\end{array}$$

BoxOffice and Friends schemas are included into EnhancedBoxOffice schema

Schema Hiding (\)

- Used to hide some variables and declarations of already known schemas (i.e., existentially quantified).
- Use ∃ to hide variables.

$$
\begin{array}{|l}
\hline
S \\
a : A \\
b : B \\
\hline
P \\
\hline
\end{array}
$$

Then S\(a) is the schema

$$
\begin{array}{|l}
\hline
b : B \\
\hline
\exists\, a : A \bullet P \\
\hline
\end{array}
$$

Schema Hiding (\)

$$
\begin{array}{|l}
\hline
\textit{A} \\
\hline
a : \mathbb{Z} \\
\hline
a = 42 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{HideA} \\
\hline
\exists\, a : \mathbb{Z} \bullet \\
\quad a = 42 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{B} \\
\hline
a, b : \mathbb{Z} \\
\hline
a = b + 2 \\
b < 10 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{HideB} \\
\hline
a : \mathbb{Z} \\
\hline
\exists\, b : \mathbb{Z} \bullet \\
\quad (a = b + 2 \land \\
\quad b < 10) \\
\hline
\end{array}
$$

- Note that $\exists\, b : \mathbb{Z} \bullet (a = b + 2 \land b < 10)$ means the same thing as $a < 12$.
- So, we can equivalently write *HideB* as:

$$
\begin{array}{|l}
\hline
\textit{HideB} \\
\hline
a : \mathbb{Z} \\
\hline
a < 12 \\
\hline
\end{array}
$$

Schema Hiding (\)

- Example:

Define $AddWho \mathrel{\widehat{=}} AddMember \setminus newMember?$:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddMember}} \\
\hline
\Delta ClubState \\
newmember? : STUDENT \\
\hline
newmember? \notin badminton \\
badminton' = badminton \cup \\
\qquad \{newmember?\} \\
hall' = hall \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddWho}} \\
\hline
\Delta ClubState \\
\hline
\exists newmember? : STUDENT \bullet \\
(newmember? \notin badminton \land \\
\ badminton' = badminton \cup \\
\quad \{newmember?\} \land \\
\ hall' = hall) \\
\hline
\end{array}
$$

Schema Composition (⨟)

- If two schemas describe operations upon the same state, then we can construct an operation schema that describes the effect of one followed by the other.
- In a schema composition, the after state of the first operation is identified with the before state of the second.
- The symbol for schema composition is fat semicolon ⨟

Schema Composition ($\fatsemi$)

1. Replace primed variables (’) in first schema with double primed variables (”)

   S[“/’]

2. Replace un-primed variables in second schema with double primed variables (”)

   T[‘/]

3. Existentially quantify variables in double primed state (“)

   ∃ State” • S[“/’] ∧ T[“/]

Schema Piping (≫)

- Constructing a new schema out of two old ones by relating the output variables of one of those old schemas to the input variables of the other one.
- For the piping S ≫ T to be defined, for each word y such that S has an output y! and T has an input y?, the types of these two components must be the same. We call x a piped variable.
- Schema are combined by schema conjunction.
- Any unmatched inputs and outputs remain in the signature of the result.

Schema Piping (>>)

- Assume we have two schemas S and T:

$$\begin{array}{|l} \hline S \\ x?, s, s', y! : \mathbb{N} \\ \hline s' = s + x? \\ y! = 2 * s' \\ \hline \end{array} \qquad \begin{array}{|l} \hline T \\ y?, t, t' : \mathbb{N} \\ \hline y? > t \\ t' = t + y? \\ \hline \end{array}$$

- We rename the same type of variables y! in S and y? in T into an entirely new variable e.g. p, so we form:

  S [p/y!]

  T [p/y?]

Schema Piping (>>)

- Assume Alpha ≙ S [p/y!] and Beta ≙ T [p/y?]

Alpha

$x?, s, s', p : \mathbb{N}$

$s' = s + x?$

$p = 2 * s'$

Beta

$p, t, t' : \mathbb{N}$

$p > t$

$t' = t + p$

- We form a conjunction of the two renamed schemas as Gamma = S [p/y!] ∧ T [p/y?]

Gamma

$x?, s, s', p, t, t' : \mathbb{N}$

$s' = s + x?$

$p = 2 * s'$

$p > t$

$t' = t + p$

Schema Piping (≫)

- And hide the variable p in Gamma, and named it as Delta: Delta = (S[p/y!] ∧ T[p/y?]) \(p)

  and this is S ≫ T

Delta

$$x?, s, s', t, t' : \mathbb{N}$$

$$\exists p : \mathbb{N} \bullet$$
$$(s' = s + x? \land$$
$$p = 2 * s' \land$$
$$p > t \land$$
$$t' = t + p)$$

Simplified version of Delta will look like this →

Delta

$$x?, s, s', t, t' : \mathbb{N}$$

$$s' = s + x?$$
$$2 * s' > t$$
$$t' = t + 2 * s'$$

Schema Implication (⇒)

- The statement "p implies q" (**$p \Rightarrow q$**) means that if p is true, then q must also be true.
- Statement p is called the premise of the implication and q is called the conclusion.

Schema Equivalence ($\Leftrightarrow$)

- The equivalence "if and only if" (**$p \Leftrightarrow q$**) means that p and q are of the same strength; thus it might also be called bi-implication.
- **$p \Leftrightarrow q$** means that both $p \Rightarrow q$ and $q \Rightarrow p$.

Schema Propositions

$\wedge$ and $\vee$ associate to the left.

$$P \wedge Q \wedge R \qquad \text{is} \qquad (P \wedge Q) \wedge R$$

$\Rightarrow$ and $\Leftrightarrow$ associate to the right.

$$P \Rightarrow Q \Rightarrow R \qquad \text{is} \qquad P \Rightarrow (Q \Rightarrow R)$$

Operators bind in the following precendence: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

$$P \wedge Q \Rightarrow \neg R \Leftrightarrow S \qquad \text{is} \qquad ((P \wedge Q) \Rightarrow (\neg R)) \Leftrightarrow S$$

Error Scenarios

Error scenarios

- Errors should be reported whenever one of the pre-conditions fail (which means the operation cannot take place)
- Schemas can be defined for each error condition.
- The final schema will combine the operator schema and the error schema condition schemas using disjunction (*or*) operators.

The Class System

- We are going to model a system for certification class system where the scenario will be as:

  Students may register to join the class providing the class is not full. The students need to successfully complete all the coursework given to get the certificate of completion.

- For this example, we are NOT going to concern with details of students such as students' number, name, date of birth etc.

The Class System

- We introduce the basic type STUDENT as a given set, the set of all students who are going to register for the class:

  [STUDENT]

- There is a limit to the number of students who can join the class. We define *maxClassSize*, the maximum number of students that can be in the class. Assume the size is 20.

$$
\begin{array}{|l}
maxClassSize : \mathbb{N} \\
\hline
maxClassSize = 20
\end{array}
$$

The Class System

- We are going to model two sets of students:

enrolled	the set of students who have joined the class
passed	a subset of enrolled and represents the set of enrolled students who have passed their courseworks

- The size of the class is constrained by *maxClassSize*.
- The two sets are declared as finite set.

The Class System

- We define the state space schema called Class:

$$
\begin{array}{|l}
\textit{Class} \\
\hline
\textit{enrolled} : \mathbb{F}\,\textit{STUDENT} \\
\textit{passed} : \mathbb{F}\,\textit{STUDENT} \\
\hline
\#\textit{enrolled} \leqslant \textit{maxClassSize} \\
\textit{passed} \subseteq \textit{enrolled} \\
\hline
\end{array}
$$

The Class System

- To begin with, there are no enrolled students and no student has passed.

$$
\begin{array}{|l}
\textit{InitClass} \\
\hline
\textit{Class} \\
\hline
\textit{enrolled} = \varnothing \\
\textit{passed} = \varnothing \\
\hline
\end{array}
$$

The Class System

- A student may join the class if the class is not already full and if the student has not already enrolled.
- A new student cannot passed all their coursework yet.

$$
\begin{array}{|l}
\hline
\textit{EnrolOk} \\
\Delta \textit{Class} \\
\textit{student?} : \textit{STUDENT} \\
\hline
\#\textit{enrolled} < \textit{maxClassSize} \\
\textit{student?} \notin \textit{enrolled} \\
\textit{enrolled}' = \textit{enrolled} \cup \{ \textit{student?} \} \\
\textit{passed}' = \textit{passed} \\
\hline
\end{array}
$$

The Class System

- An existing student is transferred to the *passed* provided they have passed all their coursework and have not already been transferred.
- Every student in *passed* must also be in *enrolled*.

$$
\begin{array}{|l}
\hline
CompleteOk \\
\Delta Class \\
student? : STUDENT \\
\hline
student? \in enrolled \\
student? \notin passed \\
enrolled' = enrolled \\
passed' = passed \cup \{ student? \} \\
\hline
\end{array}
$$

The Class System

- Only those existing students who have passed may leave with a certificate. Their records will be removed from *enrolled* and *passed*.

$$
\begin{array}{|l}
\hline
\textit{LeaveWithCertificateOk} \\
\Delta Class \\
student? : STUDENT \\
\hline
student? \in passed \\
enrolled' = enrolled \setminus \{ student? \} \\
passed' = passed \setminus \{ student? \} \\
\hline
\end{array}
$$

The Class System

- Previous *Class* system only concerned with simple, straightforward, no problem scenarios.
  - We did not concern ourselves with the possibility that the class is full and so no one can enrolled on it.
  - We ignored the possibility that a student could enrolled twice.
  - We also ignored the possibility that a student who is not enrolled could transferred to the passed set.
- We have to address these error scenarios.

The Class System

- Let’s draw up a table of pre-conditions, the set of all states for which successful outcomes are defined.

Schema	Pre-conditions for success
EnrolOk	**$\#enrolled < maxClassSize$** **$student? \notin enrolled$**
CompleteOk	**$student? \in enrolled$** **$student? \notin passed$**
LeaveWithCetificateOk	**$student? \in passed$**

The Class System

- We add on the table the conditions for failure.

Schema	Pre-conditions for failure
EnrolOk	Class is full: **$\#enrolled \geq maxClassSize$** Student already enrolled: **$student? \in enrolled$**
CompleteOk	Student not enrolled: **$student? \notin enrolled$** Student already passed: **$student? \in passed$**
LeaveWithCetificateOk	Student not passed: **$student? \notin passed$**

The Class System

1) First step of error scenarios:

Define a *free type definition* to list all the errors for the system including one for the success schema.

REPORT ::= success | classFull | alreadyEnrolled | notEnrolled | alreadyPassed | notPassed

OR

REPORT ::= success | class_full | already_enrolled | not_enrolled | already_passed | not_passed

The Class System

2) Second step, create the schema for successful scenario (e.g **Success**) where we just have one declaration and one predicate.

$$
\begin{array}{|l}
\textit{Success} \\
\hline
report! : REPORT \\
\hline
report! = success \\
\hline
\end{array}
$$

OR using text form

Success == [report!: REPORT | report! = success]

The Class System

3) Third step is to create all the schemas for failures for each identified error case. For an error case, there will be no update to any of system variables.

- The class is full if the number of enrolled students has reached the maximum class size.

$$
\begin{array}{|l}
\textit{ClassFull} \\
\hline
\Xi Class \\
report! : REPORT \\
\hline
\#enrolled \geq maxClassSize \\
report! = classFull \\
\hline
\end{array}
$$

The Class System

- A student cannot be enrolled again if they are already enrolled.

$$
\begin{array}{|l}
\textit{AlreadyEnrolled} \\
\hline
\Xi Class \\
student? : STUDENT \\
report! : REPORT \\
\hline
student? \in enrolled \\
report! = alreadyEnrolled
\end{array}
$$

The Class System

- If a student is not enrolled they cannot passed.

$$
\begin{array}{|l}
\hline
\textit{NotEnrolled} \\
\Xi Class \\
student? : STUDENT \\
report! : REPORT \\
\hline
student? \notin enrolled \\
report! = notEnrolled \\
\hline
\end{array}
$$

The Class System

- The same student cannot be passed twice.

$$
\begin{array}{|l}
\hline
\textit{AlreadyPassed} \\
\Xi \textit{Class} \\
\textit{student?} : \textit{STUDENT} \\
\textit{report!} : \textit{REPORT} \\
\hline
\textit{student?} \in \textit{passed} \\
\textit{report!} = \textit{alreadyPassed} \\
\hline
\end{array}
$$

The Class System

- A student who has not passed cannot leave with a certificate.

$$
\begin{array}{|l}
\hline
\textit{NotPassed} \\
\Xi Class \\
student? : STUDENT \\
report! : REPORT \\
\hline
student? \notin passed \\
report! = notPassed \\
\hline
\end{array}
$$

Complete Schema

Combining Schema

- When building large schemas from smaller ones such as:
  - 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
  - …

  all these need to be combine to create a complete schema for those operations.
- So, conjunction and disjunction are most useful to combine schemas.

Combining Schema

- Suppose we have schema A, B, B':

$$\begin{array}{|l}\hline A \\ a : \mathbb{Z} \\ \hline a = 42 \\ \hline\end{array} \qquad \begin{array}{|l}\hline B \\ a, b : \mathbb{Z} \\ \hline a = b + 2 \\ b < 10 \\ \hline\end{array} \qquad \begin{array}{|l}\hline B' \\ B : \mathbb{P}\mathbb{Z} \\ \hline 42 \in B \\ \hline\end{array}$$

So: AandB ≜ A ^ B

$$\begin{array}{|l}\hline AandB \\ a, b : \mathbb{Z} \\ \hline a = 42 \land (a = (b + 2) \land b < 10) \\ \hline\end{array}$$

Combining Schema – The Class System

- We use conjunction (and) to combine two schemas (successful operation schema and the success handling schema). Say **EnrolOk** and **Success**

  EnrolOk ∧ Success

- We use disjunction (or) to represent alternatives (All the error handling schemas). Say **ClassFull**, or **AlreadyEnrolled**.

  ClassFull ∨ AlreadyEnrolled

Combining Schema – The Class System

- The combination of success and failure will be by using disjunction:

$$(EnrolOk \land Success) \lor ClassFull \lor AlreadyEnrolled$$

- To create a complete schema **of the enrol process**.

$$Enrol \triangleq (EnrolOk \land Success) \lor ClassFull \lor AlreadyEnrolled$$

$\triangleq$ stands for schema definition.

Combining Schema – The Class System

- Similarly,

$$Complete \triangleq (CompleteOk \land Success) \lor NotEnrolled \lor AlreadyPassed$$

$$LeaveWithCertificate \triangleq (LeaveWithCertificateOk \land Success) \lor NotPassed$$

Complete Schema for Enrol

- **Enrol ≙ (EnrolOk ∧ Success) ∨ ClassFull ∨ AlreadyEnrolled**

$$
\begin{array}{|l}
\textit{Enrol} \\
\hline
enrolled, enrolled' : \mathbb{F}\ STUDENT \\
passed, passed' : \mathbb{F}\ STUDENT \\
student? : STUDENT \\
report! : REPORT \\
\hline
(\#enrolled < maxClassSize \land student? \notin enrolled \land enrolled' = enrolled \cup \{student?\} \land \\
passed' = passed \land report! = success \land passed \subseteq enrolled \land passed' \subseteq enrolled') \\
\lor \\
(\#enrolled \geqslant maxClassSize \land report! = classFull \land passed \subseteq enrolled \land passed' \subseteq enrolled') \\
\lor \\
(student? \in enrolled \land report! = alreadyEnrolled \land passed \subseteq enrolled \land passed' \subseteq enrolled' \\
\land \#enrolled \leqslant maxClassSize) \\
\hline
\end{array}
$$

Complete Schema for Complete

- **Complete ≙ (CompleteOk ∧ Success) ∨ NotEnrolled ∨ AlreadyPassed**

Complete

$enrolled, enrolled' : \mathbb{F}\ STUDENT$

$passed, passed' : \mathbb{F}\ STUDENT$

$student? : STUDENT$

$report! : REPORT$

$(student? \in enrolled \land student \notin passed \land enrolled' = enrolled \land passed' = passed \land$

$report! = success \land passed \subseteq enrolled \land passed' \subseteq enrolled' \land \#enrolled \leqslant maxClassSize)$

$\lor$

$(student? \notin enrolled \land report! = notEnrolled \land enrolled' = enrolled \land passed' = passed \land$

$passed \subseteq enrolled \land passed' \subseteq enrolled' \land \#enrolled \leqslant maxClassSize)$

$\lor$

$(student \in passed \land report! = alreadyPassed \land enrolled' = enrolled \land passed' = passed \land$

$passed \subseteq enrolled \land passed' \subseteq enrolled' \land \#enrolled \leqslant maxClassSize)$

Complete Schema for LeaveWithCertificate

- **LeaveWithCertificate ≙ (LeaveWithCertificateOk ∧ Success) ∨ NotPassed**

$$\begin{array}{l}
_LeaveWithCertificate____________________ \\
\mid enrolled, enrolled' : \mathbb{F}\ STUDENT \\
\mid passed, passed' : \mathbb{F}\ STUDENT \\
\mid student? : STUDENT \\
\mid report! : REPORT \\
\hline
\mid (student? \in passed \land enrolled' = enrolled \setminus \{student?\} \land passed' = passed \setminus \{student?\} \land \\
\mid report! = success \land passed \subseteq enrolled \land passed' \subseteq enrolled' \land \#enrolled \leqslant maxClassSize) \\
\mid \lor \\
\mid (student? \notin passed \land report! = notPassed \land enrolled' = enrolled \land passed' = passed \land \\
\mid passed \subseteq enrolled \land passed' \subseteq enrolled' \land \#enrolled \leqslant maxClassSize) \\
\hline
\end{array}$$

Example

[NAME, HEIGHT, WEIGHT]

$$
\begin{array}{|l}
\hline
\textit{HeightAndWeight} \\
\hline
known\_height : \mathbb{P}\ NAME \\
known\_weight : \mathbb{P}\ NAME \\
height : NAME \pfun HEIGHT \\
weight : NAME \pfun WEIGHT \\
\hline
known\_weight = \mathrm{dom}\ weight \\
known\_height = \mathrm{dom}\ height \\
\hline
\end{array}
$$

Example

$$\textit{NewHeight}$$

$$\Delta\ \textit{HeightAndWeight}$$
$$name? : NAME$$
$$hgt? : HEIGHT$$

$$name? \notin known\_height$$
$$known\_height' = known\_height \cup \{name?\}$$
$$height' = height \cup \{name? \mapsto hgt?\}$$
$$weight' = weight$$
$$known\_weight' = known\_weight$$

$$\textit{FindWeight}$$

$$\Xi\ \textit{HeightAndWeight}$$
$$name? : NAME$$
$$wgt! : WEIGHT$$

$$name? \in known\_weight$$
$$name? \in \mathrm{dom}\ weight$$
$$wgt! = weight\ name?$$
$$known\_height' = known\_height$$
$$height' = height$$

Example

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{WhoIsTallAndHeavy}} \\
\hline
\Xi\ \textit{HeightAndWeight} \\
hgt? : \textit{HEIGHT} \\
wgt? : \textit{WEIGHT} \\
names! : \mathbb{P}\ \textit{NAME} \\
\hline
names! = \{n{:}known\_height \mid height\ n = hgt?\} \cap \\
\qquad\qquad \{n{:}known\_weight \mid weight\ n = wgt?\} \\
\hline
\end{array}
$$

Example

REPORT::= ok | height_already_known | height_not_known | weight_already_known | weight_not_known

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Success}} \\
\hline
rep! : REPORT \\
\hline
rep! = ok \\
\hline
\end{array}
$$

Example

$$\begin{array}{|l}
\multicolumn{1}{l}{_HeightAlreadyKnown________________} \\
\Xi\ HeightAndWeight \\
name? : NAME \\
rep! : REPORT \\
\hline
name? \in known_height \\
rep! = height_already_known \\
\hline
\end{array}$$

(NewHeight ∧ Success) ∨ HeightAlreadyKnown

A full definition is:

FullNewHeight ≙ (NewHeight ∧ Success) ∨ HeightAlreadyKnown

Example

$$
\begin{array}{|l}
\hline
\textit{FullNewHeight} \\
\hline
known\_height, known\_height' : \mathbb{P}\ NAME \\
known\_weight, known\_weight' : \mathbb{P}\ NAME \\
height, height' : NAME \pfun HEIGHT \\
weight, weight' : NAME \pfun WEIGHT \\
name? : NAME \\
hgt? : HEIGHT \\
rep! : REPORT \\
\hline
(name? \notin known\_height \land known\_height' = known\_height \cup \{name?\} \land \\
height' = height \cup \{name? \mapsto hgt?\} \land weight' = weight \land known\_weight' = known\_weight \land \\
known\_weight = \mathrm{dom}\ weight \land known\_height = \mathrm{dom}\ height \land \\
known\_weight' = \mathrm{dom}\ weight' \land known\_height' = \mathrm{dom}\ height' \land rep! = ok) \\
\lor \\
(name? \in known\_height \land rep! = height\_already\_known \land weight' = weight \land \\
height' = height \land known\_weight' = known\_weight \land known\_height = \mathrm{dom}\ height \land \\
known\_weight = \mathrm{dom}\ weight \land known\_height = \mathrm{dom}\ height \land \\
known\_weight' = \mathrm{dom}\ weight' \land known\_height' = \mathrm{dom}\ height') \\
\hline
\end{array}
$$

Error Scenarios Exercise

Question 1

- Referring to the scenario from Chapter 2 where we want **to keep the customer information** entering a shop for one whole day.

- We were given a state space schema called **CountCustomer,** an axiomatic definition for the maximum and a basic type [CUSTOMER]

$$
\begin{array}{|l}
maximum : \mathbb{N} \\
\hline
maximum = 100
\end{array}
$$

$$
\begin{array}{|l}
\textit{CountCustomer} \\
\hline
customer : \mathbb{P}\ CUSTOMER \\
\hline
\#customer \leqslant maximum \\
\hline
\end{array}
$$

Question 1

- And three operations schema called *AddCustomer, UpdateCustomer* and *RemoveCustomer* that update the *CountCustomer* system state variable.

$$
\begin{array}{|l}
\hline
\textit{AddCustomer} \\
\hline
\Delta \textit{CountCustomer} \\
\textit{cust?} : \textit{CUSTOMER} \\
\hline
\textit{cust?} \notin \textit{customer} \\
\#\textit{customer} < \textit{maximum} \\
\textit{customer}' = \textit{customer} \cup \{\textit{cust?}\} \\
\hline
\end{array}
$$

Question 1

$$
\begin{array}{|l}
\hline
\textit{UpdateCustomer} \\
\Delta CountCustomer \\
cust? : CUSTOMER \\
\hline
cust? \in customer \\
customer' = customer \oplus \{cust?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RemoveCustomer} \\
\Delta CountCustomer \\
cust? : CUSTOMER \\
\hline
cust? \in customer \\
customer' = customer \setminus \{cust?\} \\
\hline
\end{array}
$$

Question 1

- Lets create a table:

Schema Name	Success Pre-conditions	Failure Pre-conditions	Remark
AddCustomer	cust? ∉ customer #customer < maximum	cust? ∈ customer #customer ≥ maximum	Already exist Already full
UpdateCustomer	cust? ∈ customer	cust? ∉ customer	Not exist
RemoveCustomer	cust? ∈ customer	cust? ∉ customer	Not exist

Question 1

- Introduce a free type REPORT that has two errors handling. The error handling reports are *ok, alreadyExist, notExist,* and *alreadyFull*.

REPORT::= ok | alreadyExist | notExist | alreadyFull

Question 1

- Write a schema Success that will provide an error report (rep!) for every successful error handling schema.

Question 1

- Write a schema AlreadyExist that will provide an error report (rep!) for a customer (cust?) who is already enter the shop.

Question 1

- Write a schema NotExist that will provide an error report (rep!) for a customer (cust?) who is not in the shop.

Question 1

- Write a schema AlreadyFull that will provide an error report (rep!) for a customer (cust?) who cannot enter the shop due to the maximum capacity.

Question 1

- Complete the whole schema for the adding the customer in the shop as a complete schema/total function called AddCustomerComplete.

Question 1

- Complete the whole schema for the updating the customer in the shop as a complete schema/total function called UpdateCustomerComplete.

Question 1

- Complete the whole schema for removing the customer in the shop as a complete schema/total function called RemoveCustomerComplete.

Question 2

- Referring to the scenario from Chapter 2 where we are recording the passengers aboard an aircraft. There are no seat numbers and passengers are allowed aboard on a first-come-first-served basis

$$\begin{array}{|l}\hline \textit{Aircraft} \\ \hline onboard : \mathbb{P}\ PERSON \\ \hline \#onboard \leqslant capacity \\ \hline \end{array}$$

Question 2

- We have two operation schemas **BoardAircraft** and **DisembarkAircraft**

BoardAircraft

$\Delta\ Aircraft$

$p?: PERSON$

$p? \notin onboard$

$\#onboard < capacity$

$onboard' = onboard \cup \{p?\}$

DisembarkAircraft

$\Delta\ Aircraft$

$p?: PERSON$

$p? \in onboard$

$onboard' = onboard \setminus \{p?\}$

Question 2

- Lets create a table:

Schema Name	Success Pre-conditions	Failure Pre-conditions	Remark
BoardAircraft	$p? \notin onboard$ $\#onboard < capacity$	$p? \in onboard$ $\#onboard \geq capacity$	Already onboard Full capacity
DisembarkAircraft	$p? \in onboard$	$p? \notin onboard$	Not onboard

Question 2

- Introduce a free type RESPONSE that has two errors handling. The error handling reports are *success, alreadyOnboard, notOnboard,* and *alreadyFull*

RESPONSE ::= success | alreadyOnboard | notOnboard | alreadyFull

Question 2

- Write a schema ***BoardError*** that will provide response *rep!* if it is not possible to board the aircraft.
- Assume that a successful schema ***Success*** has already been created. Complete the boarding operation as *Board*

Question 2

- Write a schema ***DisembarkError*** that will provide response *rep!* if it is not possible to disembark the aircraft.
- Assume that a successful schema ***Success*** has already been created. Complete the disembarking operation as *Disembark.*

Summary

- This lecture described in details the Z schema operators such as conjunction, disjunction, negation, composition and others.
- It also discussed about the error scenarios to handle failed pre-conditions by discussing the Class System example.
- At the end of the lecture, the schemas are combined to produce complete schemas using schema calculus.

THANK YOU!!