SCHEMA CALCULUS

This page is intentionally blank

☺

INTRODUCTION

- The *schema calculus* allows us to regard schemas as individual units which may be manipulated using certain notational conventions and operators

- Some of the ideas of schema calculus have already been encountered
  - schema inclusion,
  - schema conjunction,
  - schema decoration,
  - Δ convention, etc

  (and used without comment)

- Schema calculus allows developers to
  - structure specifications
  - combine existing schemas
  - avoid large, monolithic, unstructured schema formats

SCHEMA COMPATIBILITY

- To be able to combine schemas
  - the *signatures* of the schemas must be *type compatible*
  - *global variables* need to be considered to avoid *name clashes*
  - the resulting combination should be *meaningful*

- *Type compatibility* means
  - if the schemas to be combined have any common variable in their signatures, then that variable must have the same type (though not necessarily the same constraint) in each signature

SCHEMA COMPATIBILITY

- Consider the following schemas:

$$Schema\_A \triangleq [a:\mathbb{N}; b:1..8 \mid a = 63 \textbf{ mod } b]$$

$$Schema\_B \triangleq [b:\mathbb{N}_1; c : \text{DATE} \mid b \leq \text{month\_of } c \wedge c \text{ is\_after start\_of } 96]$$

$$Schema\_C \triangleq [b, c : \text{DATE}; d:\mathbb{N}_1]$$

$$Schema\_D \triangleq [b:\mathbb{Z}; c, d : \text{DATE} \mid b < 2*a \wedge c \text{ is\_on\_or\_before } d]$$

- *Schema_A* and *Schema_B* are type compatible because their signatures have the common variable *b* and *b* has the same type (*viz.* $\mathbb{Z}$)

- *Schema_B* and *Schema_C* share variables *b* and *c* but are not type compatible (because the *b* variables are not of the same type)

RENAMING

- *Schema_D* is type compatible with *Schema_B* but NOT with *Schema_A* because the variables named *a* are different (one is global; one local)

- Name clashes should be avoided by careful naming but the *renaming operation* can help to overcome both name clashing and also type incompatibility:

$$Schema\_Name\ [new\_var_1 / old\_var_1,\ new\_var_2 / old\_var_2,\ ...]$$

- $Schema\_A[b/a, a/b]$ causes no problems since all renaming is considered simultaneous

RENAMING

- Writing *Schema_B*[*x*/*b*] overcomes *Schema_B*'s incompatibility with *Schema_C* (renaming the clashing component from *Schema_C* would be equally effective)

- *Schema_A*[*x*/*a*] overcomes the local/global name clash between *Schema_A* and *Schema_D* (NB renaming in *Schema_D* would **not** be effective)

- Obviously, *text editing* could be used to change the name of the global variable *a* but then the entire specification is affected

RENAMING

- It must be borne in mind that *renaming* is about renaming only *components* declared for a schema and **not** all occurrences of apparently the same variable

- In particular, *bound* variables in the predicate part are not schema components and should not be affected by *renaming*

- If T is the schema:

$$
\begin{array}{|l}
\hline
T \\
x : \mathbb{Z} \\
s : \mathbb{P}\ \mathbb{Z} \\
\hline
x \in s \\
\exists\, x : \mathbb{P}\ \mathbb{Z} \bullet x \subset s \\
\hline
\end{array}
$$

then $T\,[\,y \,/\, x\,]$ is:

$$
\begin{array}{|l}
\hline
T \\
y : \mathbb{Z} \\
s : \mathbb{P}\ \mathbb{Z} \\
\hline
y \in s \\
\exists\, x : \mathbb{P}\ \mathbb{Z} \bullet x \subset s \\
\hline
\end{array}
$$

SCHEMAS AS TYPES

- The *signature* of a schema is a loose collection of variables and their types
- The *predicate* of the schema introduces a property that constrains the values that can be taken in combination
- The schema therefore defines a collection of possible values often called *bindings*
- For: $Schema\_X \triangleq [a, b : \mathbb{N} \mid a < b \land b < 3]$ there are only three possible combinations or *bindings*:

$$a \rightarrow 0; \quad b \rightarrow 1$$
$$a \rightarrow 0; \quad b \rightarrow 2$$
$$a \rightarrow 1; \quad b \rightarrow 2$$

- In Z, a collection of values is a type and the type of a schema simply describes a collection of bindings that satisfy its property

SCHEMAS AS TYPES

- Since a schema can represent a type, we must be able to declare a variable whose type is a schema binding

- By declaring: $u : Schema\_X$

  u is defined to be a variable which can have as its possible values the valid bindings of the schema $Schema\_X$ with components a and b where, at all times, $a < b \land b < 3$

- The selection operator, **.** (a *dot*), allows us to specify particular components

  - e.g. to refer to the **current** values of the a and b components of the variable u (see above), we write $u \,.\, a$ and $u \,.\, b$ - this is similar to the invocation of an object feature in *Eiffel*

BINDING FORMATION

- We may refer to the **current** binding value of a schema, S, say, when no particular variable is specified, with the notation θ S (called *binding formation*)

- θ S represents the binding of schema S that has **all** component values equal to the current values of the variables in the schema S

- Consider a small *Order Processing System* which accepts orders for a particular product

- Two types of order exist:

  - *normal order* — which is only subject to the constraint that the number of items required must not exceed the number of items currently in stock

  - *small order* — characterized by the additional constraint that the order value does not exceed a particular limit

BINDING FORMATION

- For given sets: [MONEY, ORDER_NOS]

  we declare suitable globals:

$$max\_items : \mathbb{N} \qquad \mid \; max\_items \leq 500$$

$$small\_limit : \text{MONEY} \qquad \mid \; small\_limit = 1000$$

Then, if *Order* is the schema:

Order

$$order\_no : \text{ORDER\_NOS}$$
$$qty : \mathbb{N}$$
$$item\_price : \text{MONEY}$$
$$\mid \; qty \leq max\_items$$

we can define a function *value* (which computes the value of a given order) by:

$$value : Order \rightarrow \text{MONEY} \qquad \mid \; \forall\, ord : Order \bullet value = ord.qty * ord.item\_price$$

BINDING FORMATION

- Small orders may then be defined using schema inclusion:

$$\begin{array}{|l}
\textit{Small_Order} \\
\hline
\textit{Order} \\
\hline
\textit{qty} * \textit{item_price} \leq \textit{small_limit} \\
\hline
\end{array}$$

- Note that the computation defined in *Small_Order* is exactly the same as that in the function *value* and, therefore, an alternative (and some would argue more elegant and appealing) definition for this schema is:

$$\begin{array}{|l}
\textit{Small_Order} \\
\hline
\textit{Order} \\
\hline
\textit{value}\ (\theta\ \textit{Order}) \leq \textit{small_limit} \\
\hline
\end{array}$$

where *function application* has been used

SCHEMA LOGICAL OPERATORS

- The simplest combining operators in the schema calculus are the *propositional (logical) operators* $\wedge$, $\vee$, $\neg$, $\Leftrightarrow$ and $\Rightarrow$

- If *Opr* represents one of these operators, then the combination schema produced by:

  $$\text{Schema}_1 \; \mathit{Opr} \; \text{Schema}_2$$

  comprises

  - a *declaration part* which is some merge, dependent on the nature of *Opr*, of the separate declaration parts with no multiple instances of declarations of common variables

  - a *predicate* (or *axiom*) *part* which is some combination, again dependent on *Opr*, of the separate predicate parts

SCHEMA LOGICAL OPERATORS

- Schema calculus *overloads* the logical operators but the symbolism is appropriate because of the influence that the operators have on the predicate parts of the schemas

- Consider from before:

$$Schema\_A \triangleq [a:\mathbb{N}; b:1..8 \mid a = 63 \textbf{ mod } b]$$

$$Schema\_B \triangleq [b:\mathbb{N}_1; c : \text{DATE} \mid b \leq \text{month\_of } c \land c \text{ is\_after start\_of } 96]$$

- Then if $Schema\_AorB \triangleq Schema\_A \lor Schema\_B$ the text of $Schema\_AorB$ is equivalent to:

$$[a:\mathbb{N}; b:\mathbb{Z}; c:\text{DATE} \mid (a = 63 \textbf{ mod } b \land b \in 1..8) \lor (b \geq 1 \land b \leq \text{month\_of } c \land c \text{ is\_after start\_of } 96)]$$

- $Schema\_AorB$'s property is satisfied when either $Schema\_A$'s property is satisfied, or $Schema\_B$'s, or both

SCHEMA LOGICAL OPERATORS

- More precisely, the *bindings* which satisfy *Schema_AorB* are either:
  - those which satisfy *Schema_A* when they are restricted to the signature of *Schema_A*, or
  - those which satisfy *Schema_B* when they are restricted to the signature of *Schema_B*, or
  - those which, when appropriately restricted, satisfy both schemas

- To illustrate, assuming "start_of 96" is (1,1,1996) where the date tuple has form (*day*, *month*, *year*), then each of $binding_1$, $binding_2$ and $binding_3$ satisfy *Schema_AorB*:

$binding_1$: $a \rightarrow 0$; $b \rightarrow 7$; $c \rightarrow (7, 3, 1949)$
$binding_2$: $a \rightarrow 7$; $b \rightarrow 5$; $c \rightarrow (25, 11, 1997)$
$binding_3$: $a \rightarrow 7$; $b \rightarrow 8$; $c \rightarrow (31, 8, 1996)$

SCHEMA LOGICAL OPERATORS

- Restricting $binding_1$ to *Schema_A*, yields binding:

  $$a \rightarrow 0; b \rightarrow 7$$

  which satisfies *Schema_A*'s property

- Restricting $binding_1$ to *Schema_B*, yields binding:

  $$b \rightarrow 7; \quad c \rightarrow (7, 3, 1949)$$

  which does **not** satisfy *Schema_B*'s property

- Overall, $binding_1$ satisfies *Schema_AorB*'s property

- Similar checks can be made with bindings $binding_2$ and $binding_3$

SCHEMA LOGICAL OPERATORS

- The conjunction of *Schema_A* and *Schema_B*:

  $$Schema\_AandB \triangleq Schema\_A \land Schema\_B$$

  has text equivalent to:

  $$[a:\mathbb{N}; b:\mathbb{Z}; c:\text{DATE} \mid (a = 63 \textbf{ mod } b \land b \in 1..8) \land (b \geq 1 \land b \leq \text{month\_of } c \land c \text{ is\_after start\_of } 96)]$$

- The *bindings* which satisfy *Schema_AandB* must
  - satisfy *Schema_A* when restricted to the signature of *Schema_A*, AND also
  - satisfy *Schema_B* when restricted to the signature of *Schema_B*

- Proceeding as before we find that only binding $binding_3$ satisfies *Schema_AandB*

- **Including** *Schema_B* in *Schema_A* yields identical text to *Schema_AandB*
  - hence *schema inclusion* is a special form of *schema conjunction*

SCHEMA LOGICAL OPERATORS

- The only other propositional operator much used in the schema calculus is ¬ (negation)

- In general, for a schema S, ¬ (S) has the same signature as S but a property that is the **negation** of the *property* of S (i.e. ¬ S defines a set of bindings that do **not** satisfy S)

- Care is needed in applying negation:
  - if: $Schema\_A \triangleq [a:\mathbb{N}; b:1..8 \mid a = 63 \bmod b]$ then ¬ $Schema\_A$ is **not** $[a:\mathbb{N}; b:1..8 \mid \neg(a = 63 \bmod b)]$, because
  - part of the property of $Schema\_A$ is given by the constraint in the *declaration* of b

- ¬ $Schema\_A$ is actually:
  $[a:\mathbb{Z}; b:\mathbb{Z} \mid \neg (a \geq 0 \wedge b \in 1..8 \wedge a = 63 \bmod b)]$

SCHEMA LOGICAL OPERATORS

- To further emphasise the pitfalls involved with negating schemas, consider:

$$Numbers \triangleq [\, x : \mathbb{N}_1 \mid x < 100 \,]$$

- If we negate *Numbers* we might well expect to obtain a specification which restricts x to values which are greater than or equal to 100; and might even write:

$$Negated\_Numbers \triangleq [\, x : \mathbb{N}_1 \mid x \geq 100 \,]$$

- However, *Negated_Numbers* does not attempt to negate the extra predicate *implied* in the original declaration of x

SCHEMA LOGICAL OPERATORS

- If we normalise (i.e. make both *types* and *constraints* explicit) then we recover:

  $Normalised\_Numbers \triangleq [\, x : \mathbb{Z} \mid x \geq 1 \land x < 100 \,]$

- Negating this gives:

  $\neg\, Normalised\_Numbers \triangleq [\, x : \mathbb{Z} \mid \neg\, (x \geq 1 \land x < 100)]$

- Applying De Morgan's law:

  $\neg\, Normalised\_Numbers \triangleq [x : \mathbb{Z} \mid x < 1 \lor x \geq 100)]$

  and this permits values of x which are negative!

SCHEMA LOGICAL OPERATORS

- A further warning to beware of *negation* is provided by:

$$Bounded\_Function \triangleq [f : X \pfun Y \mid \#(\mathbf{dom}\, f) \leq max]$$

- If $Not\_Bounded\_Function \triangleq \neg Bounded\_Function$

  we might suppose *Not_Bounded_Function* would mean "some function from X to Y with domain size exceeding *max*"

- But, as with *Numbers*, we are getting more than we bargained for

- Normalising and then negating yields:

$$[f : \mathbb{P}\,(X \times Y) \mid \neg\,(\,\#(\mathbf{dom}\, f) \leq max \land f \in X \pfun Y)]$$

  or

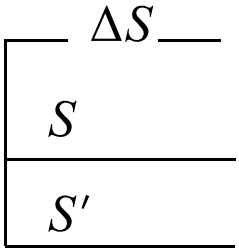
$$[f : \mathbb{P}\,(X \times Y) \mid \#(\mathbf{dom}\, f) > max \lor f \notin X \pfun Y)]$$

- This yields states which are not even functional!

THE DELTA (Δ) CONVENTION

- The Δ convention allows us to include, within one schema declaration, schemas which describe, for any operation, the *before* and *after* system states

- Conventionally, for any schema S, the schema ΔS is taken to represent:



  where S' represents the "after" state for the schema S

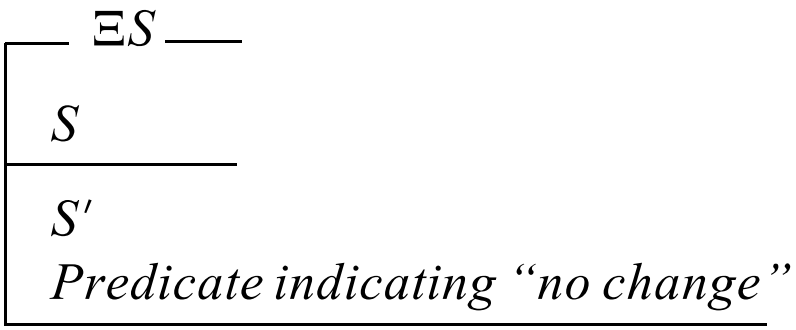
- Since ΔS represents a schema, it may be included within another schema (we have frequently exploited this idea!)

- Alternatively, we can write: $\Delta S \triangleq S \land S'$

THE XI (Ξ) CONVENTION

- The use of the Ξ symbol indicates that the schema to which it is applied suffers no change of state because of an operation

- Conventionally, for any schema S, the schema ΞS is taken to represent:



- Note that the "no change" predicate could be written as $\theta S' = \theta S$

- We could also write: $\Xi S \triangleq [\, \Delta S \mid \theta S' = \theta S \,]$

ERROR HANDLING

- The commonest use of the $\wedge$ and $\vee$ operators considered earlier is in the construction of "total" operations

- *Total* operations allow for the violation of the constraints

- We have already used the $\wedge$ and $\vee$ operators for the handling of errors in the previously issued *Simple System Specification Using Z* which was concerned with passengers boarding and disembarking from an aircraft

- We shall now revisit error handling with a partial specification from a different system (the PROGRAMMER-PROJECT System)

PROGRAMMER-PROJECT SYSTEM

- Suppose a system monitors the allocation of programmers to various projects

- Each programmer is uniquely identified by a programmer number (*prog_num*) from which the programmer's name can be derived

- Each project similarly has a unique project code (*proj_code*)

- *Assignments* are determined by a combination of programmer number and project code and reveal which programmers are engaged upon which projects

PROGRAMMER-PROJECT SYSTEM

- To be able to define a state schema for this system we must first define a suitable *state space*

- Suppose the basic types are:

[PROG_NUM]	the set of all possible programmer numbers
[PROG_NAME]	the set of all possible programmer names
[PROJ_CODE]	the set of all possible project codes
[PROJ_NAME]	the set of all possible project names

- Then useful relationships might be:

$$programmer : PROG\_NUM \nrightarrow PROG\_NAME$$
$$project : PROJ\_CODE \rightarrowtail PROG\_NAME$$
$$assignment : PROG\_NUM \leftrightarrow PROJ\_CODE$$

where $\rightarrowtail$ represents an ***injective*** (1:1) partial function

PROGRAMMER-PROJECT SYSTEM

- The state schema might then be *Project_Programmer* as shown:

Project_Programmer

$$
\begin{array}{|l}
\textit{programmer} : \text{PROG\_NUM} \pfun \text{PROG\_NAME} \\
\textit{project} : \text{PROJ\_CODE} \pinj \text{PROJ\_NAME} \\
\textit{assignment} : \text{PROG\_NUM} \rel \text{PROJ\_CODE} \\
\hline
\textbf{dom}\ \textit{assignment} \subseteq \textbf{dom}\ \textit{programmer} \\
\textbf{ran}\ \textit{assignment} \subseteq \textbf{dom}\ \textit{project} \\
\hline
\end{array}
$$

- The system constraint(s) expressed in the schema are, in essence, concisely expressible as:
  - only known programmers are assigned to known projects

PROGRAMMER-PROJECT SYSTEM

- Adding a new programmer to the system may be expressed by the schema *Add_Prog*:

Add_Prog

Δ *Project_Programmer*
progname? : PROG_NAME
prognum? : PROG_NUM

---

dom *programmer* $\cap$ { *prognum*? } = { }
programmer′ =
programmer $\oplus$ {(*prognum*?, *progname*?)}
project′ = *project*
assignment′ = *assignment*

  - But the above schema does not allow for the precondition being violated

ERROR HANDLING

- A simple device for handling system errors is to incorporate a *reporting* mechanism into the schemas

- This may be achieved by defining an appropriate *free type*, FEEDBACK, say:

  FEEDBACK ::=
  OK | *UnknownProgrammer* |
  Unknown Project | *AlreadyAssigned* ...

- Then if *Success* is the report schema:

  $Success \triangleq [report! : \text{FEEDBACK} \mid report! = OK]$

  we may define a schema, *Add_Prog_OK* to describe "normal" behaviour, as:

  $$Add\_Prog\_OK \triangleq Add\_Prog \wedge Success$$

ERROR HANDLING

- To cater for precondition violation we define a schema *Add_Prog_Error*

- If we can assume that the precondition violation does not alter the state of the system, we have:

Add_Prog_Error

Ξ *Project_Programmer*
prognum? : PROG_NUM
report! : FEEDBACK

$\neg$ (**dom** *programmer* $\cap$ { *prognum*? } = { })
report! = *AlreadyKnown*

- And the "total" operation could therefore be:

Total_Add_Prog $\triangleq$ *Add_Prog_OK* $\vee$ *Add_Prog_Error*
or
Total_Add_Prog $\triangleq$
(*Add_Prog* $\wedge$ *Success*) $\vee$ *Add_Prog_Error*

ERROR HANDLING

- If an error condition is detected a system may
  - permit other operations to be performed, or
  - remember the error and enter an undefined state which allows no operations except for *re-initialise*, or
  - preserve values and not permit other operations to be performed until some form of *error recovery* has been performed

ERROR HANDLING

- In some situations certain error conditions may be far more significant than others

- Consider an on-line multi-user computer system which allows registered users to be attached (actually connected or queued awaiting connection)

- Obvious constraints govern the attachment of a user:

  - the computer must be online
  - the user must be registered
  - there must be spare capacity on the machine (the wait queue must have space)
  - the user must not already be attached (assumes more than one concurrent attachment not to be permitted)

ERROR HANDLING

- If the computer is offline then there is no point checking for spare capacity

- If the user is not registered then a check on existing attachment is immaterial

- What if the computer is offline AND the user is not registered?
  - It may be acceptable for the system to report ambiguously or for it to report either error
  - If a definite priority is necessary then it may be advantageous to specify that "valid-user" is checked only when the computer is known to be online (see *Cerberus* case study)

SCHEMA HIDING

- Consider again the *Add_Prog* schema:

Add_Prog

Δ *Project_Programmer*
progname? : PROG_NAME
prognum? : PROG_NUM

---

dom *programmer* $\cap$ { *prognum*? } = { }
programmer′ =
programmer $\oplus$ {(*prognum*?, *progname*?)}
project′ = *project*
assignment′ = *assignment*

- The above schema requires *two* inputs

- Suppose, instead, we reduce the input effort by, more realistically, allowing the system to allocate an unused number to the new programmer name

SCHEMA HIDING

- The following schema shows the alternative approach (allowing the system to allocate the programmer number):

New_Add_Prog

$$\Delta\, Project\_Programmer$$
$$progname? : PROG\_NAME$$

$$\exists\, prognum : PROG\_NUM \bullet$$
$$(\ \mathbf{dom}\ programmer \cap \{\ prognum\ \} = \{\ \}$$
$$\wedge\ programmer' =$$
$$programmer \oplus \{(prognum, progname?)\})$$
$$project' = project$$
$$assignment' = assignment$$

- This technique can be used whenever we wish a system to allocate values rather than accept them as input

- In Z we can use *Schema Hiding* to write:

$$New\_Add\_Prog \triangleq Add\_Prog \setminus (prognum?)$$

SCHEMA COMPOSITION

- Suppose Q and R are schemas describing operations, then the expression

$$P \triangleq Q \fatsemi R$$

defines a new schema P such that if Q causes a state change from S_1 to S_2 and R can bring about a change from S_2 to S_3, then P can bring about a change from S_1 to S_3

- In particular
  - the *before* state variables of Q become the *before* state variables of P
  - the *after* state variables of Q become the *before* state variables of R
  - the *after* state variables of R become the *after* state variables of P
  - the *property* of the new schema, P, is equivalent to the properties of Q *and* R

SCHEMA COMPOSITION

- Suppose Q and R are the schemas:

$$\begin{array}{|l} Q \\ \hline x?, s, s', y! : \mathbb{N} \\ \hline s' = s - x? \\ y! = s \\ \hline \end{array} \qquad \begin{array}{|l} R \\ \hline x?, s, s' : \mathbb{N} \\ \hline s < x? \\ s' = s \\ \hline \end{array}$$

- To form Q ⨾ R:
  - rename the *after* state variables in Q to something new, s^+, say (i.e. form $Q[s^+/s']$)
  - rename the *before* state variables in R to the same new thing, s^+ (i.e. form $R[s^+/s]$)
  - form the conjunction: $Q[s^+/s'] \land R[s^+/s]$
  - hide the new variable, s^+, introduced previously

SCHEMA COMPOSITION

- Suppose $Q_1 = Q[s^+/s']$ and $R_1 = R[s^+/s]$:

Q_1

$x?, s, s^+, y! : \mathbb{N}$

$s^+ = s - x?$

$y! = s$

R_1

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

$s^+ < x?$

$s' = s^+$

- Then, if: $Q_1\_and\_R_1 \triangleq Q[s^+/s'] \land R[s^+/s]$:

$Q_1\_and\_R_1$

$x?, s, s^+, s', y! : \mathbb{N}$

$s^+ = s - x?$

$y! = s$

$s^+ < x?$

$s' = s^+$

- The next step is to hide the new variable, s^+, to recover a first draft, $(Q \fatsemi R)_0$, say:

$$(Q \fatsemi R)_0 \triangleq (Q[s^+/s'] \land R[s^+/s]) \setminus (s^+)$$

SCHEMA COMPOSITION

- Hence we have:

$$(Q \fatsemi R)_0$$

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

$$\exists s^+ : \mathbb{N} \bullet (s^+ = s - x? \land y! = s \land s^+ < x? \land s' = s^+)$$

- Although the above schema represents the composition of the schemas Q and R it is expressed in terms of the introduced variable s^+

- Simplifying (remembering s^+ is the *after state* in Q and the *before state* in R) we recover Q ⨾ R as:

$$Q \fatsemi R$$

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

$$s' = s - x?$$
$$y! = s$$
$$s' < x?$$

SUMMARY OF SYMBOLS

$S[x_1/y_1, \ldots, x_n/y_n]$	*the variables* $y_1, \ldots, y_n$, *declared in schema S, are renamed as variables* $x_1, \ldots, x_n$ *respectively*
$v \,.\, p$	*gives current value of the component p where p is a component declared in schema S and v is a variable of a type S*
θS	*represents the binding of schema S that has component values equal to the current values of the variables in S*
S_1 *opr* S_2	*yields a schema derived from schemas* S_1 *and* S_2 *according to which propositional operator is used for "opr"*
$S \setminus (a, \ldots)$	*hiding of components from schema S*
$S_1 \fatsemi S_2$	*sequential composition of schemas* S_1 *and* S_2

This page is intentionally blank

EXERCISES

1. Say which *pairs* of schemas could not be combined, as they stand, and explain why (you may assume variables to be global if not explicitly declared):

$S_1 \triangleq [\, c, d, e : 0 \,..\, 9 \,]$
$S_2 \triangleq [\, b, x? : \text{DATE} \mid b \neq x? \,]$
$S_3 \triangleq [a : \mathbb{N}; d : \mathbb{N}_1 \mid d - a = c \,]$
$S_4 \triangleq [\, a, b, x! : \mathbb{Z}; c' : \text{DATE} \mid a < 0 \land b \neq 0 \land (\, day\_of\ c' = -a \textbf{ mod } b \,]$

2. Write out the schema text which is equivalent to the following expression:

$$S_4 [\, a / b,\ b / c',\ c' / a \,]$$

where S_4 is the schema defined in the previous question.

3. Given the schema definitions:

$S_5 \triangleq [\, x, y : \mathbb{N} \mid x < n \land y < m \,]$, and
$S_6 \triangleq [\, \Delta S_5 \mid x' = y \land y' = x \,]$

write out, exploiting the usual Z conventions, the texts of:

(a) $S_6 \lor \Xi S_5$
(b) $\neg \Delta S_5$

4. Suppose S and T are the schemas given by:

$S \triangleq [\, x : \mathbb{Z}; y : \mathbb{N} \mid x > y \,]$
$T \triangleq [\, y : \mathbb{Z}; s : \mathbb{P}\, \mathbb{Z} \mid y \in s \land \exists x : \mathbb{P}\, \mathbb{Z} \bullet x \subset s \,]$

Write out the schemas

(a) *S_and_T* where *S_and_T* $\triangleq S \land T$
(b) *S_or_T* where *S_or_T* $\triangleq S \lor T$

5. Consider the schema alongside which describes an investment strategy that specifies investment in a company with a profit of at least one million pounds:

Very_Profitable_Company

c : COMPANY
$x : \mathbb{N}$

---

profit $c = x$
$x \geq 1000000$

Suppose there is a complete turnabout so that the investment strategy should specify investment in companies that do *not* make a profit of at least one million pounds.

Demonstrate that the negation of *Very_Profitable_Company* will **not** yield an appropriate schema.

6. Consider the following schema which attempts to describe the instruction to buy shares in a given company provided the share price does not exceed a given value:

Buy_Subject_To_Price

$c? : \text{COMPANY}$
$price? : \mathbb{N}$
$action! : \text{ACTION}$

$share\_price\ c? \leq price?$
$action! = buy$

There are occasions when the expectation is that share prices are bound to rise (a *bull* market) and investors will buy shares at any price. If *Buy_At_Any_Price* is the schema corresponding to the bull market situation we might write:

$$Buy\_At\_Any\_Price \triangleq Buy\_Subject\_To\_Price \setminus (price?)$$

Write out the schema description for *Buy_At_Any_Price*.

7. The schema for *Buy_At_Any_Price* should contain an existential quantifier ($\exists$). If $\exists$ was replaced by the universal quantifier ($\forall$) what, in English, would the modified schema mean?

8. In the notes on the *Applications of Sequences* one of the examples dealt with *stacks*. At the end a comment was made that in a correct stack specification the sequential composition of operations *Push* and *Pop* should yield an unchanged stack. Demonstrate that this is, in fact, the case.

9. Suppose an organization has an internal telephone network. A database is created which holds details of people and associated extension number(s). Some people may have more than one extension and some extensions may be shared by more than one person; so, we might have:

[PERSON]	the set of all people who might part of the database at some time
[PHONE]	the set of all possible telephone extension numbers that might be allocated at some time

If we use *telephones* to act as an identifier expressing the relation between people and their extension numbers, we have (from the comments above):

$$telephones : PERSON \leftrightarrow PHONE$$

The state of the database might be specified by:

Phone_DB

$$members : \mathbb{P}\ PERSON \quad \text{-- } \textit{set of all people in the organization}$$
$$telephones : PERSON \leftrightarrow PHONE$$

$$\textbf{dom}\ telephones \subseteq members$$

(a) Specify schemas *Add_Entry* and *Remove_Entry* to define the operations of, respectively, **adding** and **deleting** an entry from the database.

(b) Use schema composition to show how the operation of someone **changing** their extension can be composed from the *Remove_Entry* and *Add_Entry* operations.