The Z specification language

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University Montreal, Canada

February 1, 2022

Introduction

- Z (pronounced "zed") is a *specification language* based on predicate logic and set theory.
- A formal specification in Z describes *what* the system does, without saying *how* it does it.
- The formal specification is implementation independent.

Example: Birthday book

- In this example we are defining the specification of a birthday book.
- We introduce the set of all names, and the set of all dates, as basic types in the specification:

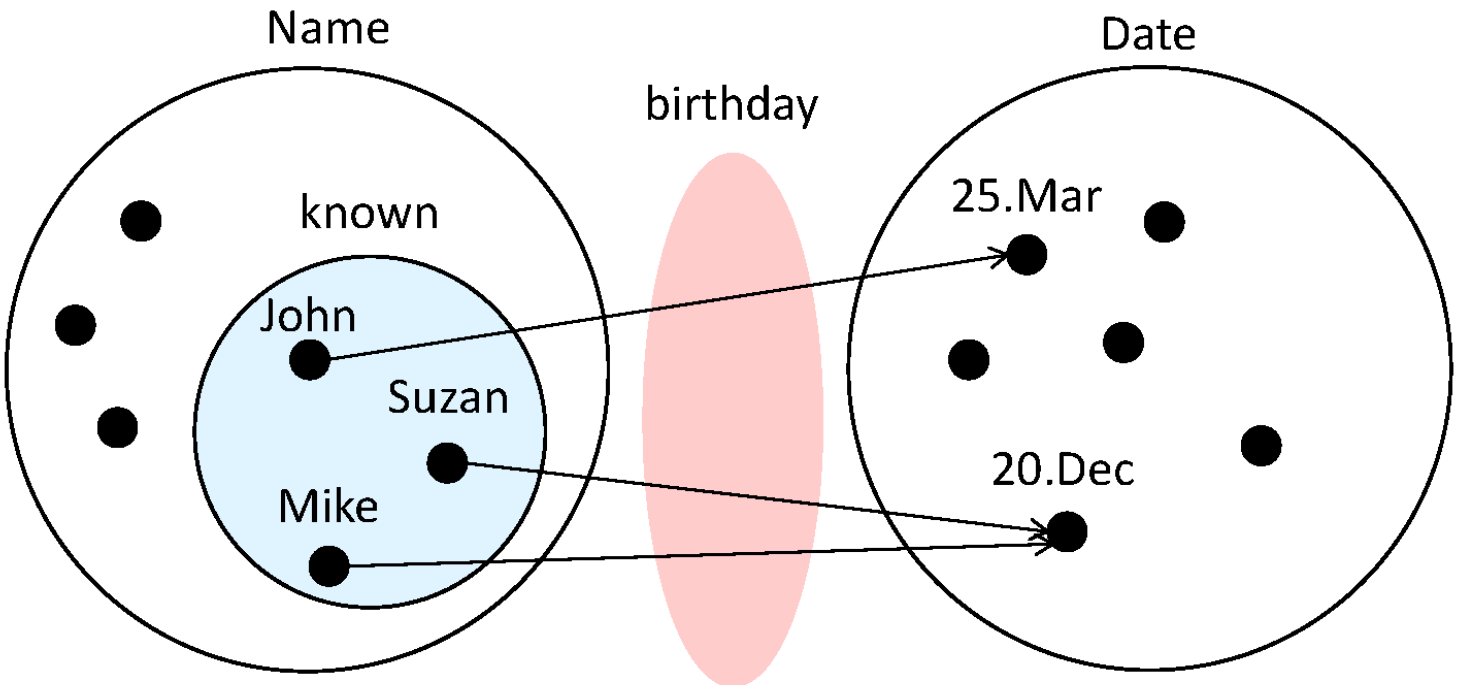
$$[Name, Date]$$

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

$$known = \{John, Mike, Suzan\}$$

$$birthday = \{ John \mapsto 25.Mar, Mike \mapsto 20.Dec, Suzan \mapsto 20.Dec \}$$

Birthday book /cont. - Function *birthday*



Birthday book: State schema

- A specification is decomposed into *schemata* (plural for *schema*).
- A Z *schema* can provide a high-level description of a model.

BirthdayBook

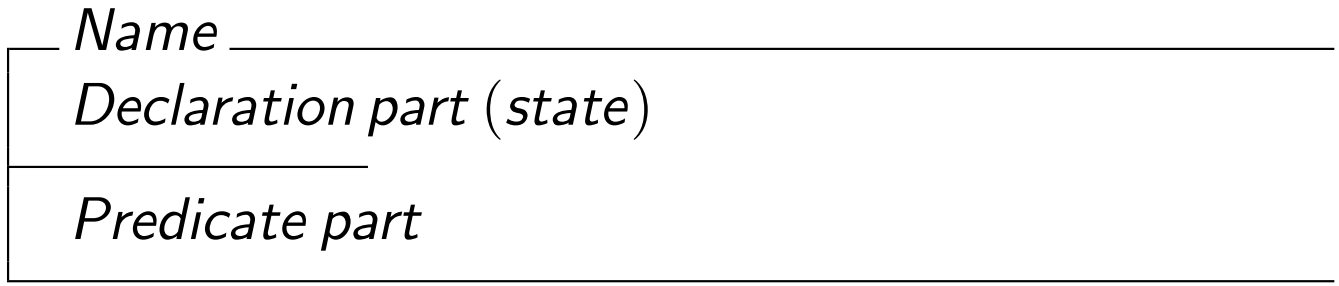
$$known : \mathbb{P}\, Name$$
$$birthday : Name \pfun Date$$

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

- Variable *known* is the set of names with birthdates recorded and *birthday* is a function which, when applied to certain names, gives the birthdays associated with them.
- Note that *birthday* is a *partial function*: It does not relate every element of *Name* to *Date*.

Schemata /cont.

- We define a schema for both the data of the model as well as for each of the system's operations.



- A schema consists of
- **Name**
- **Declaration part (state)**: specifies a set of variable assignments and their types (called the *state* of the system).
- **Predicate part**
  - Properties of the model expressed in terms of the variables defined in the declaration part.
  - These properties constitute *state invariants* (must be true at all times).

Schemata /cont.

- Alternatively a schema can be expressed in a linear notation as

$$name \mathrel{\hat{=}} [declaration\ part \mid predicate\ part]$$

Decorations: Before and after state

- *Decorations* give identifiers conventional meanings.
- A variable decorated with the symbol ' denotes its value after having being manipulated by an operation.
- Thus, *known* and *birthday* denote the states of their corresponding sets before an operation is invoked, whereas $known'$ and $birthday'$ denote the states of their corresponding sets after an operation has been successfully terminated.

A birthday book /cont.

- The predicate $known = \mathrm{dom}\ birthday$ states that the set *known* is the same as the domain of the function *birthday*, i.e. the set of names that the function can apply.
- This relationship is an *invariant* of the system.

BirthdayBook

$$known : \mathbb{P}\, Name$$
$$birthday : Name \pfun Date$$

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

A birthday book: State schema

- In our schema we have managed to capture the information that

1. Each person can have only one birthday (because *birthday* is a function), and
2. Two (or more) people can share the same birthday (as in the example).

- The invariant $\mathit{known} = \mathrm{dom}\ \mathit{birthday}$ is satisfied, because *birthday* records a date for exactly the three names in set *known*.

A birthday book: Operation *AddBirthday*

- Once the state space is defined, we must provide a schema for each of the operations of the system.
- Operation *AddBirthday* adds a new birthday into the system:

AddBirthday

$$\Delta BirthdayBook$$
$$name? : Name$$
$$date? : Date$$

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

- The declaration $\Delta BirthdayBook$ captures the fact that the schema describes a *state change*: it introduces variables $known$, $birthday$, and $known'$, $birthday'$.
- The first is an observation of the state before the change, and the last two are observations of the state after the change.

A birthday book: Operation *AddBirthday*

- Next come the declarations of the two inputs to the operation and by convention, the names of inputs end in a question mark.

$$name? : Name$$
$$date? : Date$$

A birthday book: Operation *AddBirthday*

- The first line under the dividing line defines the precondition to the operation:

  $$name? \notin known$$

- The name to be added must not already be one of those known to the system, as each person can have only one birthday.
- The specification can be extended to include error conditions in the case of a precondition failure.

A birthday book: Operation *AddBirthday*

- If the precondition is satisfied, the next two lines specify the postcondition to the operation:

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

1. $name?$ is now a member of the set $known$.
2. The birthday function is extended to map the new name to the given date.

A birthday book: Operation *FindBirthday*

- Operation *FindBirthday* finds the birthday of a person known to the system:

FindBirthday

$$\Xi BirthdayBook$$
$$name? : Name$$
$$date! : Date$$

$$name? \in known$$
$$date! = birthday(name?)$$

- The declaration $\Xi BirthdayBook$ indicates that this is an operation in which the state of the system does not change.
- It makes the following statements redundant:

$$known' = known$$
$$birthday' = birthday$$

Operation *FindBirthday*: Declaring an output

- The use of a name ending in an exclamation mark defines an output:

  $$date! : Date$$

Operation *FindBirthday*: Precondition and output

- The precondition of the operation is that $name?$ is one of the names known to the system:

  $$name? \in known$$

- If this is so, then the output $date!$ is the value of the birthday function given argument $name?$.

  $$date! = birthday(name?)$$

A birthday book: Operation *Remind*

- Operation *Remind* finds which people have a birthday on a given date:

Remind

$$\Xi BirthdayBook$$
$$today? : Date$$
$$cards! : \mathbb{P}\, Name$$

$$cards! = \{n : known \mid birthday(n) = today?\}$$

- This time there is no precondition.
- The output *cards*! is defined as the set of all values drawn from the set *known* such that the value of the birthday function at *n* is *today*?.

Initializing the system

- To complete the specification, we must state the initial state of the system:

InitBirthdayBook

$BirthdayBook$

$known = \emptyset$

Handling errors

- What will happen if the user tries to add a birthday for someone already known to the system? Or if they try to find the birthday of someone not known?
- Our current specification makes no claim about what should happen: The system may ignore incorrect input, or it may break down (among other things).
- In addition to the basic specification, we can describe, separately the errors which might be detected and the desired responses to them.
- We can then use operations to combine the two descriptions into a stronger (robust) specification.

Handling errors /cont.

- We define Report as an enumerated set of values that an output variable *result*! may assume:

$$Report ::= ok \mid already\_known \mid not\_known.$$

- We can now define a schema that specifies that the result should be *ok*.

Success

$$\Xi BirthdayBook$$
$$result! : Report$$

$$result! = ok$$

- We can now combine this description with *AddBirthday* as

$$AddBirthday \wedge Success$$

Handling errors /cont.

- For each precondition error we must define a schema which describes the conditions under which the error occurs and specifies the appropriate report to be produced.
- Schema *AlreadyKnown* specifies that the report *already_known* should be produced when the input *name*? is already a member of *known*:

AlreadyKnown

$$\Xi BirthdayBook$$
$$name? : Name$$
$$result! : Report$$

$$name? \in known$$
$$result! = already\_known$$

Strengthening the specification: *RAddBirthday*

- We can now provide a robust version of *AddBirthday*:

$$RAddBirthday \mathrel{\hat{=}} (AddBirthday \land Success) \oplus AlreadyKnown$$

- This definition introduces a new schema obtained by combining the schemas on the right-hand side.
- Alternatively, operation *RAddBirthday* could be specified directly by writing a single schema which combines the predicate parts of the three schemas *AddBirthday*, *Success* and *AlreadyKnown*.

Strengthening the specification: *RFindBirthday*

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

NotKnown

$$\Xi BirthdayBook$$
$$name? : Name$$
$$result! : Report$$

$$name? \notin known$$
$$result! = not\_known$$

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

$$RFindBirthday \mathrel{\hat{=}} (FindBirthday \land Success) \oplus NotKnown$$

Strengthening the specification: *RRemind*

- Operation *Remind* never results in an error, so its robust version need only add the reporting of success:

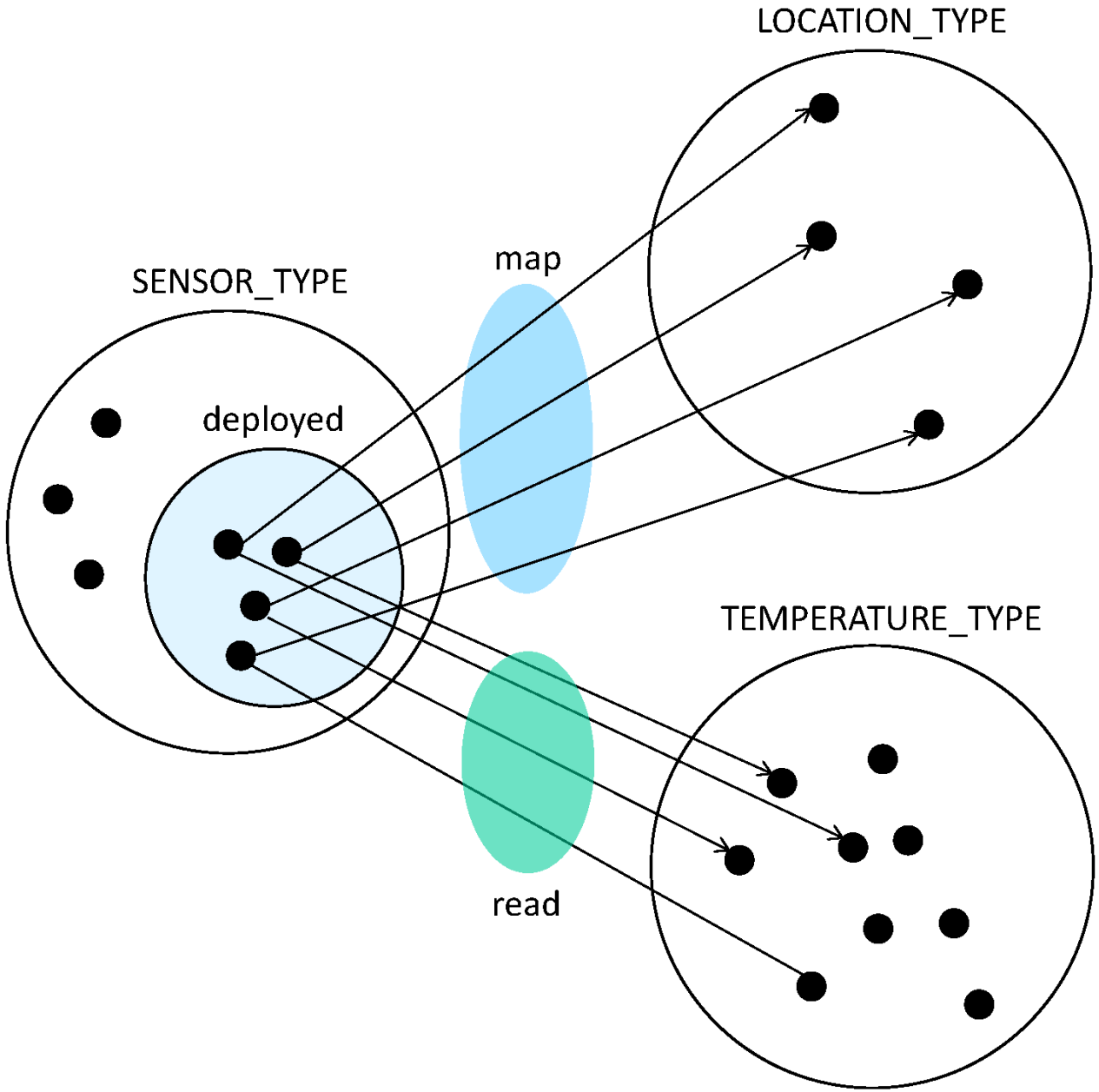
$$RRemind \mathrel{\hat{=}} Remind \wedge Success$$

Example: Temperature monitoring

- A system maintains a number of sensors, where each is deployed in a separate location in order to read the location's temperature.
- Before the system is deployed, all locations are marked on a map, and each location will be addressed by a sensor.
- The formal specification of the system introduces the following three types:

$$SENSOR\_TYPE,$$
$$LOCATION\_TYPE,$$
$$TEMPERATURE\_TYPE$$

Visualization of the system



State schema

TempMonitor

$$deployed : \mathbb{P}\, SENSOR\_TYPE$$
$$map : SENSOR\_TYPE \pfun LOCATION\_TYPE$$
$$read : SENSOR\_TYPE \pfun TEMPERATURE\_TYPE$$

$$deployed = \mathrm{dom}\, map$$
$$deployed = \mathrm{dom}\, read$$

Operation DeploySensorOK

- Operation `DeploySensorOK` places a new sensor to a unique location.
- We may assume that some (default) temperature is also passed as an argument.

DeploySensorOK

$\Delta TempMonitor$

$sensor? : SENSOR\_TYPE$

$location? : LOCATION\_TYPE$

$temperature? : TEMPERATURE\_TYPE$

$sensor? \notin deployed$

$location? \notin \operatorname{ran} map$

$deployed' = deployed \cup \{sensor?\}$

$map' = map \cup \{sensor? \mapsto location?\}$

$read' = read \cup \{sensor? \mapsto temperature?\}$

Operation ReadTemperatureOK

- Operation `ReadTemperatureOK` obtains the temperature reading from a sensor, given the sensor's location.

ReadTemperatureOK

Ξ *TempMonitor*

location? : *LOCATION_TYPE*

temperature! : *TEMPERATURE_TYPE*

---

location? $\in \operatorname{ran}$ *map*

temperature! $=$ *read*$(\mathit{map}^{-1}(\mathit{location}?))$

Success and error schemata

- We introduce an enumerated type *MESSAGE* which will assume values that correspond to success and error messages.

Success

$$\Xi TempMonitor$$
$$response! : MESSAGE$$

$$response! = 'ok'$$

Error schema SensorAlreadyDeployed for DeploySensorOK

- Operation `DeploySensorOK` places the following precondition: $sensor? \notin deployed$
- We provide an error schema for the case where this precondition fails:

$$
\begin{array}{|l}
\hline
\textit{SensorAlreadyDeployed} \\
\Xi TempMonitor \\
sensor? : SENSOR\_TYPE \\
response! : MESSAGE \\
\hline
sensor? \in deployed \\
response! = \text{'Sensor deployed'} \\
\hline
\end{array}
$$

Error schema LocationAlreadyCovered for DeploySensorOK

- Operation `DeploySensorOK` places the following precondition: $location? \notin \operatorname{ran} map$
- We provide an error schema for the case where this precondition fails:

LocationAlreadyCovered

$\Xi TempMonitor$

$location? : LOCATION\_TYPE$

$response! : MESSAGE$

$location? \in \operatorname{ran} map$

$response! = \text{'Location already covered'}$

Error schema LocationUnknown for ReadTemperatureOK

- Operation `ReadTemperatureOK` places the following precondition: $location? \in \operatorname{ran} map$
- We provide an error schema for the case where this precondition fails:

LocationUnknown

$\Xi TempMonitor$

$location? : LOCATION\_TYPE$

$response! : MESSAGE$

$location? \notin \operatorname{ran} map$

$response! = \text{'Location not covered'}$

Defining reliable operations

$$DeploySensor \mathrel{\hat{=}} (DeploySensorOK \wedge Success) \oplus (SensorAlreadyDeployed \vee LocationAlreadyCovered)$$

$$ReadTemperature \mathrel{\hat{=}} (ReadTemperatureOK \wedge Success) \oplus LocationUnknown$$

Bibliography

1. V. S. Alagar and K. Periyasamy, *Specification of Software Systems*, 2nd. ed., Springer, 2011.
2. J. Jacky *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.
3. J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd. ed., Prentice Hall International (UK) Ltd, 1992.

Appendix A: Operations on sets

Operator	Synopsis	Meaning
$\in$	$x \in S$	set membership
$\cup$	$S_1 \cup S_2$	set union
$\cap$	$S_1 \cap S_2$	set intersection
$\setminus$	$S_1 \setminus S_2$	set difference
#	#S	cardinality of a set
$\subseteq$	$S_1 \subseteq S_2$	subset
$\subset$	$S_1 \subset S_2$	proper subset
=	$S_1 = S_2$	set equality
$\bigcup$	$\bigcup$ SS	generalized union of sets SS
$\bigcap$	$\bigcap$ SS	generalized intersection of sets SS
$\mathbb{P}$	$\mathbb{P}$ S	power set of the set S
$\mathbb{F}$	$\mathbb{F}$ S	finite subsets of the set S

Appendix B: Notations for functions

Symbol	Meaning
$\rightarrow$	Total function
$\nrightarrow$	Partial function
$\rightarrowtail$	Total injective function
$\nrightarrow$	Partial injective function
$\twoheadrightarrow$	Total surjective function
$\nrightarrow$	Partial surjective function
$\rightarrowtail$	Partial bijective function
$\rightarrowtail$	Total bijective function
$\nrightarrow$	Finite partial function
$\rightarrowtail$	Finite partial injective function

Appendix C: Operations on relations and functions

Operator	Synopsis	Meaning
$\leftrightarrow$	$X \leftrightarrow Y$	declaration of a binary relation between X and Y
$\mapsto$	$x \mapsto y$	maplet
dom	dom R	domain of the relation R
ran	ran R	range of the relation R
id	id X	identity relation
$\fatsemi$	$R_1 \fatsemi R_2$	relational composition
$\circ$	$R_1 \circ R_2$	backward relational composition
$\lhd$	$S \lhd R$	domain restriction
$\rhd$	$R \rhd S$	range restriction
$\lhd$	$S \lhd R$	domain subtraction (domain anti-restriction)
$\rhd$	$R \rhd S$	range subtraction (range anti-restriction)
$\sim$	$R\sim$	relational inverse
$\_(\vert\_\vert)$	$R\ (\vert\ S\ \vert)$	relational image
$\oplus$	$R_1 \oplus R_2$	relational overriding
	R^k	relational iteration
	R^+	transitive closure of the relation R
	R^*	reflexive transitive closure of the relation R

Appendix D: Operations on sequences

Operator	Synopsis	Meaning
#	# S	length of the sequence S
$\frown$	$S_1 \frown S_2$	concatenation of sequence S_1 with S_2
rev	*rev* S	reverse of the sequence S
head	*head* S	first element of the sequence S
last	*last* S	last element of the sequence S
tail	*tail* S	sequence S with its first element removed
front	*front* S	sequence S with its last element removed
$\frown/$	$\frown/$ SS	distributed concatenation of the sequence of sequences SS
$\subseteq$	S $\subseteq$ T	S is a sequence forming the prefix of the sequence T
suffix	S suffix T	S is a sequence forming the suffix of the sequence T
in	S in T	S is a segment inside the sequence T
$\upharpoonleft$	U $\upharpoonleft$ S	extract the elements from the sequence S corresponding to the index set U; the result is also a sequence, maintaining the same order as in S
$\upharpoonright$	S $\upharpoonright$ V	extract the elements of the set V from the sequence S; the result is also a sequence, maintaining the same order as in S
disjoint	disjoint SeqSet	SeqSet is an indexed family of mutually distinct sets
partitions	SeqSet partitions T	the indexed family of mutually disjoint sets whose distributed union is T