Z Formal Specification Language

19

Keywords

Sets, relations and functions
Bags and sequences
Data reification
Refinement
Schema calculus
Proof in Z

19.1 Introduction

Z is a formal specification language based on Zermelo set theory. It was developed at the Programming Research Group at Oxford University in the early 1980s [1], and became an ISO standard in 2002. Z specifications are mathematical and employ a classical two-valued logic. The use of mathematics ensures precision, and allows inconsistencies and gaps in the specification to be identified. Theorem provers may be employed to demonstrate that the software implementation meets its specification.

Z is a '*model oriented*' approach with an explicit model of the state of an abstract machine given, and operations are defined in terms of this state. Its mathematical notation is used for formal specification, and the schema calculus is used to structure the specifications. The schema calculus is visually striking, and consists essentially of boxes, with these boxes or schemas used to describe operations and states. The schemas may be used as building blocks and combined with other

G. O'Regan, *Guide to Discrete Mathematics*, Texts in Computer Science,
DOI 10.1007/978-3-319-44561-8_19

schemas. The simple schema below (Fig. 19.1) is the specification of the positive square root of a real number.

The schema calculus is a powerful means of decomposing a specification into smaller pieces or schemas. This helps to make *Z* specifications highly readable, as each individual schema is small in size and self-contained. Exception handling is addressed by defining schemas for the exception cases. These are then combined with the original operation schema. Mathematical data types are used to model the data in a system, these data types obey mathematical laws. These laws enable simplification of expressions, and are useful with proofs.

Operations are defined in a precondition/postcondition style. A precondition must be true before the operation is executed, and the postcondition must be true after the operation has executed. The precondition is implicitly defined within the operation. Each operation has an associated proof obligation to ensure that if the precondition is true, then the operation preserves the system invariant. The system invariant is a property of the system that must be true at all times. The initial state itself is, of course, required to satisfy the system invariant.

The precondition for the specification of the square root function above is that $num? \geq 0$; i.e., the function *SqRoot* may be applied to positive real numbers only. The postcondition for the square root function is $root!^2 = num?$ and $root! \geq 0$. That is, the square root of a number is positive and its square gives the number. Postconditions employ a logical predicate which relates the pre-state to the post-state, with the post-state of a variable being distinguished by priming the variable, e.g., v'.

Z is a typed language and whenever a variable is introduced its type must be given. A type is simply a collection of objects, and there are several standard types in Z. These include the natural numbers $\mathbb{N}$, the integers $\mathbb{Z}$ and the real numbers $\mathbb{R}$. The declaration of a variable x of type X is written x: X. It is also possible to create your own types in Z.

Various conventions are employed within *Z* specification, for example $v?$ indicates that v is an input variable; $v!$ indicates that v is an output variable. The variable $num?$ is an input variable and $root!$ is an output variable for the square root example above. The notation Ξ in a schema indicates that the operation *Op* does not affect the state; whereas the notation Δ in the schema indicates that *Op* is an operation that affects the state.

SqRoot

$num?, root! : \mathbb{R}$

$num? \geq 0$

$root!^2 = num?$

$root! \geq 0$

Fig. 19.1 Specification of positive square root

Library
$on\text{-}shelf, missing, borrowed : \mathbb{P}\ Bkd\text{-}Id$
$on\text{-}shelf \cap missing = \emptyset$
$on\text{-}shelf \cap borrowed = \emptyset$
$borrowed \cap missing = \emptyset$

Fig. 19.2 Specification of a library system

Borrow
Δ *Library*
$b? : Bkd\text{-}Id$
$b? \in on\text{-}shelf$
$on\text{-}shelf' = on\text{-}shelf \setminus \{b?\}$
$borrowed' = borrowed \cup \{b?\}$

Fig. 19.3 Specification of borrow operation

Many of the data types employed in *Z* have no counterpart in standard programming languages. It is therefore important to identify and describe the concrete data structures that ultimately will represent the abstract mathematical structures. As the concrete structures may differ from the abstract, the operations on the abstract data structures may need to be refined to yield operations on the concrete data that yield equivalent results. For simple systems, direct refinement (i.e., one step from abstract specification to implementation) may be possible; in more complex systems, deferred refinement[1] is employed, where a sequence of increasingly concrete specifications are produced to yield the executable specification. There is a calculus for combining schemas to make larger specifications, and this is discussed later in the chapter.

Example 6.1 The following is a *Z* specification to borrow a book from a library system. The library is consists of books that are on the shelf; books that are borrowed; and books that are missing (Fig. 19.2). The specification models a library with sets representing books on the shelf, on loan or missing. These are three mutually disjoint subsets of the set of books *Bkd-Id*.

The system state is defined in the *Library* schema below, and operations such as *Borrow* and *Return* affect the state. The *Borrow* operation is specified in (Fig. 19.3).

The notation $\mathbb{P}$*Bkd-Id* is used to represent the power set of *Bkd-Id* (i.e., the set of all subsets of *Bkd-Id*). The disjointness condition for the library is expressed by the

[1]Step-wise refinement involves producing a sequence of increasingly more concrete specifications until eventually the executable code is produced. Each refinement step has associated proof obligations to prove that the refinement step is valid.

requirement that the pair wise intersection of the subsets *on-shelf, borrowed, missing* is the empty set.

The precondition for the *Borrow* operation is that the book must be available on the shelf to borrow. The postcondition is that the borrowed book is added to the set of borrowed books and is removed from the books on the shelf.

Z has been successfully applied in industry including the CICS project at IBM Hursley in the UK.[2] Next, we describe key parts of Z including sets, relations, functions, sequences and bags.

19.2 Sets

Sets were discussed in Chap. 2 and this section focuses on their use in Z. Sets may be enumerated by listing all of their elements. Thus, the set of all even natural numbers less than or equal to 10 is

$$\{2, 4, 6, 8, 10\}$$

Sets may be created from other sets using set comprehension: i.e., stating the properties that its members must satisfy. For example, the set of even natural numbers less than 10 is given by set comprehension as

$$\{n : \mathbb{N} | n \neq 0 \land n < 10 \land n \bmod 2 = 0 \ \cdot \ n\}$$

There are three main parts to the set comprehension above. The first part is the signature of the set and this is given by n: $\mathbb{N}$ above. The first part is separated from the second part by a vertical line. The second part is given by a predicate, and for this example the predicate is $n \neq 0 \land n < 10 \land n \bmod 2 = 0$. The second part is separated from the third part by a bullet. The third part is a term, and for this example it is simply n. The term is often a more complex expression: e.g., $\log(n^2)$.

In mathematics, there is just one empty set. However, since Z is a typed set theory, there is an empty set for each type of set. Hence, there are an infinite number of empty sets in *Z*. The empty set is written as $\varnothing$ [X] where X is the type of the empty set. In practice, X is omitted when the type is clear.

Various operations on sets such as union, intersection, set difference and symmetric difference are employed in Z. The power set of a set X is the set of all subsets of X, and is denoted by $\mathbb{P}$ X. The set of non-empty subsets of X is denoted by $\mathbb{P}_1$X where

$$\mathbb{P}_1 \mathrm{X} = \{\mathrm{U} : \mathbb{P}\,\mathrm{X} | \mathrm{U} \neq \varnothing[\mathrm{X}]\}$$

[2]This project claimed a 9 % increase in productivity attributed to the use of formal methods.

A finite set of elements of type X (denoted by $\boldsymbol{F}$ X) is a subset of X that cannot be put into a one to one correspondence with a proper subset of itself. This is defined formally as

$$F\,\mathrm{X} = \{\mathrm{U} : \mathbb{P}\mathrm{X} | \neg \exists \mathrm{V} : \mathbb{P}\mathrm{U} \cdot \mathrm{V} \neq \mathrm{U} \wedge (\exists f : \mathrm{V} \rightarrowtail \mathrm{U})\}$$

The expression $f : \mathrm{V} \rightarrowtail \mathrm{U}$ denotes that f is a bijection from U to V and injective, surjective and bijective functions were discussed in Chap. 2.

The fact that Z is a typed language means that whenever a variable is introduced (e.g., in quantification with $\forall$ and $\exists$) it is first declared. For example, $\forall j$: J $\cdot$ P $\Rightarrow$ Q. There is also the unique existential quantifier $\exists_1$ j: J | P which states that there is exactly one j of type J that has property P.

19.3 Relations

Relations are used extensively in Z and were discussed in Chap. 2. A relation R between X and Y is any subset of the Cartesian product of X and Y; i.e., R $\subseteq$ (X $\times$ Y), and a relation in Z is denoted by R: X $\leftrightarrow$ Y. The notation $x \mapsto y$ indicates that the pair $(x, y) \in$ R.

Consider, the relation *home_owner: Person* $\leftrightarrow$ *Home* that exists between people and their homes. An entry *daphne* $\mapsto$ *mandalay* $\in$ *home_owner* if *daphne* is the owner of *mandalay*. It is possible for a person to own more than one home:

$$\begin{aligned} &\mathit{rebecca} \mapsto \mathit{nirvana} \in \mathit{home\_owner} \\ &\mathit{rebecca} \mapsto \mathit{tivoli} \in \mathit{home\_owner} \end{aligned}$$

It is possible for two people to share ownership of a home:

$$\begin{aligned} &\mathit{rebecca} \mapsto \mathit{nirvana} \in \mathit{home\_owner} \\ &\mathit{lawrence} \mapsto \mathit{nirvana} \in \mathit{home\_owner} \end{aligned}$$

There may be some people who do not own a home, and there is no entry for these people in the relation *home_owner*. The type *Person* includes every possible person, and the type *Home* includes every possible home. The domain of the relation *home_owner* is given by

$$x \in \mathrm{dom}\, \mathit{home\_owner} \Leftrightarrow \exists h : \mathit{Home} \cdot x \mapsto h \in \mathit{home\_owner}.$$

The range of the relation *home_owner* is given by

$$h \in \mathrm{ran}\, \mathit{home\_owner} \Leftrightarrow \exists x : \mathit{Person} \cdot x \mapsto h \in \mathit{home\_owner}.$$

The composition of two relations *home_owner*: *Person* $\leftrightarrow$ *Home* and *home_value*: *Home* $\leftrightarrow$ *Value* yields the relation *owner_wealth*: *Person* $\leftrightarrow$ *Value* and is given by the relational composition *home_owner; home_value*
where

$$p \mapsto v \in home\_owner; home\_value \Leftrightarrow \\ (\exists h : Home \cdot \mapsto h \in home\_owner \land h \mapsto v \in home\_value)$$

The relational composition may also be expressed as

$$owner\_wealth = home\_value \circ home\_owner$$

The union of two relations often arises in practice. Suppose a new entry *aisling* $\mapsto$ *muckross* is to be added. Then this is given by

$$home\_owner' = home\_owner \cup \{aisling \mapsto muckross\}$$

Suppose that we are interested in knowing all females who are house owners. Then we restrict the relation *home_owner* so that the first element of all ordered pairs has to be female. Consider *female*: $\mathbb{P}$ *Person* with {*aisling*, *rebecca*} $\subseteq$ *female*.

$$home\_owner = \{aisling \mapsto muckross, rebecca \mapsto nirvana, \\ lawrence \mapsto nirvana\}$$

$$female \lhd home\_owner = \{aisling \mapsto muckross, rebecca \mapsto nirvana\}$$

That is, *female* $\lhd$ *home_owner* is a relation that is a subset of *home_owner*, and the first element of each ordered pair in the relation is female. The operation $\lhd$ is termed domain restriction and its fundamental property is

$$x \mapsto y \in U \lhd R \Leftrightarrow (x \in U \land x \mapsto y \in \mathrm{R}\}$$

where R: X $\leftrightarrow$Y and *U*: $\mathbb{P}$ *X*.

There is also a domain anti-restriction (subtraction) operation and its fundamental property is

$$x \mapsto y \in U \lhd\!\!\!\!- R \Leftrightarrow (x \notin U \land x \mapsto y \in \mathrm{R}\}$$

where R: X $\leftrightarrow$Y and *U*: $\mathbb{P}$*X*.

There are also range restriction (the $\rhd$ operator) and the range anti-restriction operator (the $\rhd\!\!\!\!-$ operator). These are discussed in [1].

19.4 Functions

A function [1] is an association between objects of some type X and objects of another type Y such that given an object of type X, there exists only one object in Y associated with that object. A function is a set of ordered pairs where the first element of the ordered pair has at most one element associated with it. A function is therefore a special type of relation, and a function may be *total* or *partial*.

A total function has exactly one element in Y associated with each element of X, whereas a partial function has at most one element of Y associated with each element of X (there may be elements of X that have no element of Y associated with them).

A partial function from X to Y ($f : X \pfun Y$) is a relation $f : X \leftrightarrow Y$ such that:

$$\forall x : \mathrm{X}; y, z : \mathrm{Y} \cdot (x \mapsto y \in f \land x \mapsto z \in f \Rightarrow y = z)$$

The association between x and y is denoted by $f(x) = y$, and this indicates that the value of the partial function f at x is y. A total function from X to Y (denoted f: $X \to Y$) is a partial function such that every element in X is associated with some value of Y.

$$f : X \to Y \Leftrightarrow f : X \pfun Y \land \mathrm{dom} f = X$$

Clearly, every total function is a partial function but not vice versa.

One operation that arises quite frequently in specifications is the function override operation. Consider the following specification of a temperature map:

$$\begin{array}{|l} \textit{TempMap} \\ \hline \textit{CityList} : \mathbb{P} \textit{City} \\ \textit{temp} : \textit{City} \pfun \mathbb{Z} \\ \hline \mathrm{dom}\ \textit{temp} = \textit{CityList} \\ \hline \end{array}$$

Suppose the temperature map is given by $temp = \{Cork \mapsto 17, Dublin \mapsto 19, London \mapsto 15\}$. Then consider the problem of updating the temperature map if a new temperature reading is made in Cork: e.g., $\{Cork \mapsto 18\}$. Then the new temperature chart is obtained from the old temperature chart by function override to yield $\{Cork \mapsto 18, Dublin \mapsto 19, London \mapsto 15\}$. This is written as:

$$temp' = temp \oplus \{Cork \mapsto 18\}$$

The function override operation combines two functions of the same type to give a new function of the same type. The effect of the override operation is that the entry $\{Cork \mapsto 17\}$ is removed from the temperature chart and replaced with the entry $\{Cork \mapsto 18\}$.

Suppose $f, g: X \nrightarrow Y$ are partial functions then $f \oplus g$ is defined and indicates that f is overridden by g. It is defined as follows:

$$(f \oplus g)(x) = g(x) \text{ where } x \in \text{dom}\, g$$
$$(f \oplus g)(x) = f(x) \text{ where } x \notin \text{dom}\, g \wedge x \in \text{dom}\, f$$

This may also be expressed (using domain anti-restriction) as

There is notation in Z for injective, surjective and bijective functions. An injective function is one to one: i.e.,

$$f(x) = f(y) \Rightarrow x = y.$$

A surjective function is onto: i.e.,

$$\text{Given } y \in Y, \exists x \in X \text{ such that } f(x) = y$$

A bijective function is one to one and onto, and it indicates that the sets X and Y can be put into one to one correspondence with one another. Z includes lambda calculus notation (λ-calculus was discussed in Chap. 12) to define functions. For example, the function cube = λx: $\mathbf{N} \cdot x * x * x$. Function composition $f\,;g$ is similar to relational composition.

19.5 Sequences

The type of all sequences of elements drawn from a set X is denoted by seq X. Sequences are written as $\langle x_1, x_2, \ldots . x_n \rangle$ and the empty sequence is denoted by $\langle\rangle$. Sequences may be used to specify the changing state of a variable over time, with each element of the sequence representing the value of the variable at a discrete time instance.

Sequences are functions and a sequence of elements drawn from a set X is a finite function from the set of natural numbers to X. A partial finite function f from X to Y is denoted by $f: X \nrightarrow\!\!\!\!\!\!\rightarrow Y$. A finite sequence of elements of X is given by $f: \mathbf{N} \nrightarrow\!\!\!\!\!\!\rightarrow X$, and the domain of the function consists of all numbers between 1 and # f (where #f is the cardinality of f). It is defined formally as

$$\text{seq } X == \{f: \mathbf{N} \nrightarrow\!\!\!\!\!\!\rightarrow X \mid \text{dom } f = 1 \,..\, \#f \cdot f\}$$

The sequence $\langle x_1, x_2, \ldots . x_n \rangle$ above is given by:

$$\{1 \mapsto x_1, 2 \mapsto x_2, \ldots . n \mapsto x_n\}$$

There are various functions to manipulate sequences. These include the sequence concatenation operation. Suppose $\sigma = \langle x_1, x_2, \ldots . x_n \rangle$ and $\tau = \langle y_1, y_2, \ldots . y_m \rangle$ then:

$$\sigma^\frown \tau = \langle x_1, x_2, \ldots . x_n, y_1, y_2, \ldots . y_m \rangle$$

The head of a non-empty sequence gives the first element of the sequence.

$$\text{head}\, \sigma = \text{head}\, \langle x_1, x_2, \ldots . x_n \rangle = x_1$$

The tail of a non-empty sequence is the same sequence except that the first element of the sequence is removed.

$$\text{tail}\sigma = \text{tail}\langle x_1, x_2, \ldots . x_n \rangle = \langle x_2, \ldots . x_n \rangle$$

Suppose f: $X \rightarrow Y$ and a sequence σ: seq X then the function map applies f to each element of σ:

$$\text{map}f\, \sigma = \text{map}f \langle x_1, x_2, \ldots . x_n \rangle = \langle f(x_1), f(x_2), \ldots . f(x_n) \rangle$$

The map function may also be expressed via function composition as

$$\text{map}f\, \sigma = \sigma ; f$$

The reverse order of a sequence is given by the rev function:

$$\text{rev}\, \sigma = \text{rev} \langle x_1, x_2, \ldots . x_n \rangle = \langle x_n, \ldots . x_2, x_1 \rangle$$

19.6 Bags

A bag is similar to a set except that there may be multiple occurrences of each element in the bag. A bag of elements of type X is defined as a partial function from the type of the elements of the bag to positive whole numbers. The definition of a bag of type X is

$$\text{bag}\, X = X \nrightarrow \mathbb{N}_1 .$$

For example, a bag of marbles may contain 3 blue marbles, 2 red marbles, and 1 green marble. This is denoted by B = [*b, b, b, g, r, r*]. The bag of marbles is thus denoted by

$$\text{bag}\, Marble = Marble \nrightarrow \mathbb{N}_1 .$$

$$
\begin{array}{|l}
\hline
\Delta Vending\ Machine \\
stock : \text{bag}\ Good \\
price : Good \rightarrow \mathbb{N}_1 \\
\hline
\text{dom}\ stock \subseteq \text{dom}\ price \\
\hline
\end{array}
$$

Fig. 19.4 Specification of vending machine using bags

The function count determines the number of occurrences of an element in a bag. For the example above, count *Marble b* = 3, and count *Marble y* = 0 since there are no yellow marbles in the bag. This is defined formally as

$$
\begin{array}{ll}
\text{count bag}\, X\, y = 0 & y \notin \text{bag X} \\
\text{count bag}\, X\, y = (\text{bag}\, X)(y) & y \in \text{bag X}
\end{array}
$$

An element y is in bag X if and only if y is in the domain of bag X.

$$
y \,\text{in bag}\, X \Leftrightarrow y \in \text{dom}(\text{bag}\, X)
$$

The union of two bags of marbles B_1 = [*b, b, b, g, r, r*] and B_2 = [*b, g, r, y*] is given by $B_1 \uplus B_2$ = [*b, b, b, b, g, g, r, r, r, y*]. It is defined formally as

$$
\begin{array}{lll}
(B_1 \uplus B_2)\,(y) = & B_2(y) & y \notin \text{dom}\, B_1 \wedge y \in \text{dom}\, B_2 \\
(B_1 \uplus B_2)\,(y) = & B_1(y) & y \in \text{dom}\, B_1 \wedge y \notin \text{dom}\, B_2 \\
(B_1 \uplus B_2)\,(y) = & B_1(y) + B_2(y) & y \in \text{dom}\, B_1 \wedge y \in \text{dom}\, B_2
\end{array}
$$

A bag may be used to record the number of occurrences of each product in a warehouse as part of an inventory system. It may model the number of items remaining for each product in a vending machine (Fig. 19.4).

The operation of a vending machine would require other operations such as identifying the set of acceptable coins, checking that the customer has entered sufficient coins to cover the cost of the good, returning change to the customer, and updating the quantity on hand of each good after a purchase. A more detailed examination is in [1].

19.7 Schemas and Schema Composition

The schemas in Z are visually striking and the specification is presented in two-dimensional graphic boxes. Schemas are used for specifying states and state transitions, and they employ notation to represent the before and after state (e.g., s and s' where s' represents the after state of s). They group all relevant information that belongs to a state description.

There are a number of useful schema operations such as schema inclusion, schema composition and the use of propositional connectives to link schemas together. The Δ convention indicates that the operation affects the state whereas the Ξ convention indicates that the state is not affected. These operations and conventions allow complex operations to be specified concisely, and assist with the readability of the specification. Schema composition is analogous to relational composition, and allows new schemas to be derived from existing schemas.

A schema name S_1 may be included in the declaration part of another schema S_2. The effect of the inclusion is that the declarations in S_1 are now part of S_2 and the predicates of S_1 are S_2 are joined together by conjunction. If the same variable is defined in both S_1 and S_2, then it must be of the same type in both schemas.

$$\begin{array}{|l} \text{–}S_1\text{——} \\ x,y : \mathbb{N} \\ \hline x + y > 2 \\ \hline \end{array} \qquad \begin{array}{|l} \text{–}S_2\text{——} \\ S_1 ; z : \mathbb{N} \\ \hline z = x + y \\ \hline \end{array}$$

The result is that S_2 includes the declarations and predicates of S_1 (Fig. 19.5):

Two schemas may be linked by propositional connectives such as $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \Rightarrow S_2$, and $S_1 \Leftrightarrow S_2$. The schema $S_1 \vee S_2$ is formed by merging the declaration parts of S_1 and S_2, and then combining their predicates by the logical $\vee$ operator. For example, $S = S_1 \vee S_2$ yields (Fig. 19.6):

Schema inclusion and the linking of schemas use normalization to convert sub-types to maximal types, and predicates are employed to restrict the maximal type to the sub-type. This involves replacing declarations of variables (e.g., u: 1.35 with u: Z, and adding the predicate $u > 0$ and $u < 36$ to the predicate part of the schema).

$$\begin{array}{|l} \text{–}S_2\text{——} \\ x,y : \mathbb{N} \\ z : \mathbb{N} \\ \hline x + y > 2 \\ z = x + y \\ \hline \end{array}$$

Fig. 19.5 Schema inclusion

$$\begin{array}{|l} \text{–}S\text{——} \\ x,y : \mathbb{N} \\ z : \mathbb{N} \\ \hline x + y > 2 \vee z = x + y \\ \hline \end{array}$$

Fig. 19.6 Merging schemas ($S_1 \vee S_2$)

The Δ and Ξ conventions are used extensively, and the notation Δ *TempMap* is used in the specification of schemas that involve a change of state. The notation Δ *TempMap* represents:

$$\Delta TempMap = TempMap \land TempMap'$$

The longer form of Δ *TempMap* is written as

$$\begin{array}{|l} \text{–}\Delta TempMap\text{———} \\ CityList, CityList' : \mathbb{P}\ City \\ temp, temp' : City \pfun \mathbb{Z} \\ \hline \text{dom } temp = CityList \\ \text{dom } temp' = CityList' \\ \text{———————} \end{array}$$

The notation Ξ *TempMap* is used in the specification of operations that do not involve a change to the state.

$$\begin{array}{|l} \text{–}\Xi\ TempMap\text{———} \\ \Delta TempMap \\ \hline CityList = CityList' \\ temp = temp' \\ \text{———————} \end{array}$$

Schema composition is analogous to relational composition and it allows new specifications to be built from existing ones. It allows the after state variables of one schema to be related with the before variables of another schema. The composition of two schemas S and T (S; T) is described in detail in [1] and involves 4 steps (Table 19.1):

The example below should make schema composition clearer. Consider the composition of S and T where S and T are defined as follows

$$\begin{array}{|l} \text{–S———} \\ x, x', y? : \mathbb{N} \\ \hline x' = y? - 2 \\ \text{———} \end{array} \qquad \begin{array}{|l} \text{–T———} \\ x, x' : \mathbb{N} \\ \hline x' = x + 1 \\ \text{———} \end{array}$$

$$\begin{array}{|l} \text{–}S_1\text{———} \\ x, x^+, y? : \mathbb{N} \\ \hline x^+ = y? - 2 \\ \text{———} \end{array} \qquad \begin{array}{|l} \text{–}T_1\text{———} \\ x^+, x' : \mathbb{N} \\ \hline x' = x^+ + 1 \\ \text{———} \end{array}$$

S_1 and T_1 represent the results of Step 1 and Step 2, with x' renamed to x^+ in S, and x renamed to x^+ in T. Step 3 and Step 4 yield (Fig. 19.7).:

Schema composition is useful as it allows new specifications to be created from existing ones.

Table 19.1 Schema composition

Step	Procedure
1.	Rename all *after* state variables in S to something new: S $[s^+/s']$
2.	Rename all *before* state variables in T to the same new thing: i.e., T $[s^+/s]$
3.	Form the conjunction of the two new schemas: S $[s^+/s'] \wedge$T $[s^+/s]$
4.	Hide the variable introduced in Steps 1 and 2. S; T = (S $[s^+/s'] \wedge$T $[s^+/s]) \backslash (s^+)$

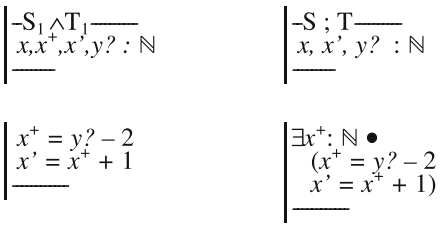

Fig. 19.7 Schema composition

19.8 Reification and Decomposition

A Z specification involves defining the state of the system and then specifying the required operations. The Z specification language employs many constructs that are not part of conventional programming languages, and a Z specification is therefore not directly executable on a computer. A programmer implements the formal specification, and mathematical proof may be employed to prove that a program meets its specification.

Often, there is a need to write an intermediate specification that is between the original Z specification and the eventual program code. This intermediate specification is more algorithmic and uses less abstract data types than the Z specification. The intermediate specification is termed the design and the design needs to be correct with respect to the specification, and the program needs to be correct with respect to the design. The design is a refinement (reification) of the state of the specification, and the operations of the specification have been decomposed into those of the design.

The representation of an abstract data type such as a set by a sequence is termed data reification, and data reification is concerned with the process of transforming an abstract data type into a concrete data type. The abstract and concrete data types are related by the retrieve function, and the retrieve function maps the concrete data type to the abstract data type. There are typically several possible concrete data types for a particular abstract data type (i.e., refinement is a relation), whereas there is one abstract data type for a concrete data type (i.e., retrieval is a function). For example, sets are often reified to unique sequences; however, more than one unique sequence can represent a set whereas a unique sequence represents exactly one set.

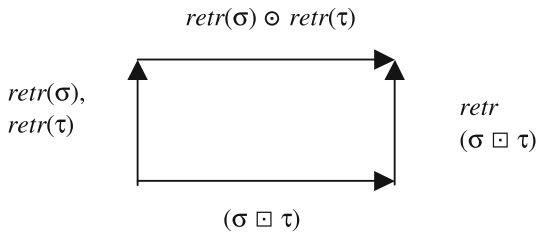

Fig. 19.8 Refinement commuting diagram

The operations defined on the concrete data type are related to the operations defined on the abstract data type. That is, the commuting diagram property is required to hold (Fig. 19.8). That is, for an operation ⊡ on the concrete data type to correctly model the operation Θ on the abstract data type the following diagram must commute, and the commuting diagram property requires proof. That is, it is required to prove that:

$$ret(\sigma \boxdot \tau) = (ret\ \sigma) \odot (ret\ \tau)$$

In Z, the refinement and decomposition is done with schemas. It is required to prove that the concrete schema is a valid refinement of the abstract schema, and this gives rise to a number of proof obligations. It needs to be proved that the initial states correspond to one another, and that each operation in the concrete schema is correct with respect to the operation in the abstract schema, and also that it is applicable (i.e., whenever the abstract operation may be performed the concrete operation may also be performed).

19.9 Proof in Z

Mathematicians perform rigorous proof of theorems using technical and natural language. Logicians employ formal proofs to prove theorems using propositional and predicate calculus. Formal proofs generally involve a long chain of reasoning with every step of the proof justified. Rigorous proofs involve precise reasoning using a mixture of natural and mathematical language. Rigorous proofs [1] have been described as being analogous to high level programming languages, whereas formal proofs are analogous to machine language.

A mathematical proof includes natural language and mathematical symbols, and often many of the tedious details of the proof are omitted. Many proofs in formal methods such as Z are concerned with crosschecking on the details of the specification, or on the validity of the refinement step, or proofs that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and tool support is essential as proof by hand often contain errors or jumps in reasoning. Machine proofs are lengthy and largely unreadable; however, they provide extra confidence as every step in the proof is justified.

The proof of various properties about the programs increases confidence in its correctness.

19.10 Review Questions

1. Describe the main features of the Z specification language.
2. Explain the difference between $\mathbb{P}_1$ X, $\mathbb{P}$ X and $\boldsymbol{F}$X.
3. Give an example of a set derived from another set using set comprehension. Explain the three main parts of set comprehension in Z.
4. Discuss the applications of Z and which areas have benefited most from their use? What problems have arisen?
5. Give examples to illustrate the use of domain and range restriction operators and domain and range anti-restriction operators with relations in Z.
6. Give examples to illustrate relational composition.
7. Explain the difference between a partial and total function, and give examples to illustrate function override.
8. Give examples to illustrate the various operations on sequences including concatenation, head, tail, map and reverse operations.
9. Give examples to illustrate the various operations on bags.
10. Discuss the nature of proof in Z and tools to support proof.
11. Explain the process of refining an abstract schema to a more concrete representation, the proof obligations that are generated, and the commuting diagram property.

19.11 Summary

Z is a formal specification language that was developed in the early 1980s at Oxford University in England. It has been employed in both industry and academia, and it was use successfully on the IBM's CICS project. Its specifications are mathematical, and this leads to more rigorous software development. Its mathematical approach allows properties to be proved about the specification, and any gaps or inconsistencies in the specification may be identified.

Z is a 'model oriented' approach and an explicit model of the state of an abstract machine is given, and the operations are defined in terms of their effect on the state. Its main features include a mathematical notation that is similar to VDM, and the schema calculus. The latter consists essentially of boxes and are used to describe operations and states.

The schema calculus enables schemas to be used as building blocks to form larger specifications. It is a powerful means of decomposing a specification into smaller pieces, and helps with the readability of *Z* specifications, as each individual schema is small in size and self-contained.

Z is a highly expressive specification language, and it includes notation for sets, functions, relations, bags, sequences, predicate calculus and schema calculus. Z specifications are not directly executable as many of its data types and constructs are not part of modern programming languages. Therefore, there is a need to refine the Z specification into a more concrete representation, and prove that the refinement is valid.

Reference

1. Z. An Introduction to Formal Methods. Antoni Diller. John Wiley and Sons. England. 1990.