Chapter 4

Z and Set-based specification

4.1 Overview

The specification language Z (pronounced “Zed”) has goals similar to those of Larch, but based on the model of set theory. In Z, everything is a set. Sets, with the usual operations of union, intersection, and membership, familiar from high-school math and introductory Discrete Math courses, are pushed to the limit and augmented by adding new notations, so that realistic systems can be specified. In Z very brief specifications convey large amounts of information. Because it is rich in notation and unfamiliar symbols, the language is difficult to present both briefly and understandably. To the uninitiated, a Z specification can appear daunting.

Some of the design goals for Z resemble those of Larch, in particular, the possibility of building up a complex specification in stages. However, some of those stages differ. In Z it is natural to separate the declarations and creation of the state from any invariant properties of states, and separate those from any dynamic operations. In Larch, it will be recalled, it was natural to add functionality to operations in stages, and to define their interrelationships in different stages. Z has a highly developed calculus for combining objects of the language, and also for refining an abstract specification into one that is closer to an
implementation. On the other hand, there is no concept of an interface language, and the connection between a Z specification and a code-level implementation is not treated within the Z methodology.

Particular effort has been invested by the developers of Z, at the Oxford Programming Research Group, in applying the language to complex examples. Some of these have been specified in cooperation with industry, and demonstrate an impressive variety of application areas. Several books and tutorials on the language have been prepared, as part of the extensive presentation effort of the developers. Z also has its own Use-net group, with over 20,000 regular readers worldwide.

Somewhat less emphasis has been put on tools that support Z. There are macros that support writing in Z style within Latex, and simple tools that enable type checking, syntax checks, and cross references. However, there is no serious theorem proving effort to semiautomatically justify specifications or refinements. Thus the implications of a specification need to be analyzed by hand.

Below major parts of the notation are presented briefly. The luxury of a gradual exposure to the notation in a book-length presentation cannot be employed here, so most of the motivation and use of the symbols will be postponed to several larger examples later in the chapter.

### 4.2 Schemas and some notation for sets

Z has one notation for declaring all objects: types, variables, functions, and operations are all instances of *schemas*. A schema syntactically consists of a variable declaration part and a predicate. A schema $S$ may be written

$$S = [declarations \mid predicate]$$

Alternatively, a two dimensional version may be used, for clarity and emphasis. This version is written as

```
S
| declarations
| predicate
```
Whenever a schema $S$ has been defined, the notation $w : S$ can be used to indicate an instance of $S$ to be called $w$, roughly that $w$ is an object of type $S$, and this is the form of declarations in the first part of a schema. If $x$ appears in the declaration part of $S$, and $w$ is an instance of $S$, then the $x$-th component of $w$ is denoted either as $w_x$ or $x(w)$.

For the predicate part, a restriction on the variables from the declaration is given in a variant of first-order logic. Standard logical connectives such as $\land$, $\lor$, or implication ($\Rightarrow$) can be used. Existential quantification is denoted by

$$\exists x : T \bullet P$$

This is the assertion “there exists an $x$ of type $T$ that satisfies $P$”

Within a schema, a standard set notation is used. Thus,

$$\{ x : T \mid P \}$$

denotes “the set that includes all $x$ of type $T$ that satisfy $P$.” One convenient extension of this notation allows defining sets of complex objects. When $a \bullet$ appears to the right within a set definition, it will be followed by an expression defining the elements composing the set. Thus,

$$\{ x : T \mid P \bullet r \}$$

means “the set of possible values of the expression $r$ such that given the declaration $x : T$, $P$ holds.” For example,

$$\{ x : Integer \mid 1 \leq x \leq 4 \bullet x^2 \}$$

is the set $\{1, 8, 27, 64\}$.

A few other common set notations include $\mathbb{P}S$, the power set of $S$, namely, the set of all subsets of $S$. If $S = \{a, b, c\}$, then the power set of $S$ is $\{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. A separate notation, $\mathbb{F}S$ denotes the finite subsets of $S$, even if $S$ itself is infinite. $\#S$ denotes the number of elements in a finite set $S$, and is undefined for infinite sets.
4.3 Relations and functions

Before a specification can be made using even a fragment of Z, the notation for relations and functions must be introduced. As noted in every introductory course in set theory, these can be viewed as special cases of sets. A relation is nothing more than a set of ordered tuples. In all of the continuation, we shall consider only binary relations, so that a relation $R$ can be described by a set of ordered pairs. If $R$ is a relation between elements of a set $X$ and elements of a set $Y$, we can use the Cartesian product notation $X \times Y$ to denote all possible pairs \{$(x, y)$\}, and thus $R : \mathcal{P}(X \times Y)$, and the pair $(x, y)$ is in the set $R$ if $x$ and $y$ satisfy the relation. Special notation used for relations includes $xRy$ instead of $(x, y) \in R$, $R : X \leftrightarrow Y$ instead of $R : \mathcal{P}(X \times Y)$, $x \mapsto y$ (read as "$x$ maps to $y$") instead of $(x, y)$.

If we consider the $fs$ (father-son) relation for Biblical personalities, then $(\text{Abraham, Isaac})$, Isaac $\mapsto$ Jacob, Jacob $fs$ Joseph are all ways of describing part of the relation.

We can describe the composition of relations directly in terms of their set representation. If a relation $R : X \leftrightarrow Y$ and a relation $S : Y \leftrightarrow Z$ are composed, the result is a relation $T : X \leftrightarrow Z$ that is defined only when the second component of $R$ coincides with the first component of $S$. In notation already presented, it is the set

$$\{x : X, y : Y, z : Z \mid x \mapsto y \in R \land y \mapsto z \in S \cdot x \mapsto z\}$$

This operation on relations is denoted by $\circ$. If a relation brothers had been defined previously, then the relation uncle-nephew contains the composition of brothers and $fs$, written brothers $\circ fs$. When a relation $R$ is composed with itself, this is denoted as $R^2$. The grandfather-grandson relation clearly contains $fs^2$, but in general could have additional entries (through a daughter who is also a mother).

If we consider a relation from a ‘source’ $X$ to a ‘target’ $Y$ (so $R : X \leftrightarrow Y$), then define

$$\text{dom } R = \{x : X \mid \exists y : Y \cdot (x, y) \in R\}$$
That is, the domain of $R$ is the set of those elements of $X$ actually related by $R$ to at least one element of $Y$.

Similarly, define

$$\text{ran } R = \{ y : Y \mid \exists x : X \bullet (x, y) \in R \}$$

The range of $R$ is the set of elements of $Y$ related by $R$ to some element of $X$.

Note that unless a relation is symmetric (as for brothers), the order of the elements in the pairs is important. For any relation $R$, the relation $R^\sim$ is obtained by reversing the order of every pair in $R$. A less common operator that restricts a relation $R$ to a set $S$ is defined by

$$S \triangleleft R = \{ x : X : y : Y \mid x \in S \land x \mapsto y \in R \bullet x \mapsto y \}$$

This is the part of $R$ that ‘starts’ in $S$, more precisely, the pairs from $R$ with a first component in $S \cap \text{dom } R$. If $R = \{(0, 1), (1, 1), (1, 0), (0, 2)\}$, then $\{0\} \triangleleft R$ is $\{(0,1), (0,2)\}$. The related operator $\triangleleft$ restricts $R$ to those pairs with domain elements not in the set $S$. Similarly, $R \triangleright S$ is the part of $R$ with range elements in $S$.

A partial function is simply a relation for which each domain element relates to exactly one range element. Z provides a rich variety of arrow symbols to express special types of functions, both in declarations and in the predicates of schemas. In Table 4.1, these arrows and their meaning are summarized. The child-mother relation would most naturally be a partial function $cm : A \rightarrow B$ from a group of people $A$ to a group of people $B$, assuming we mean the biological mother (otherwise, because of step-mothers, it is a relation but not necessarily a function). It is partial because the mother of some member of $A$ might not be in $B$. If the ages of all members of a group $G$ are given, this is a total function from $G$ to the natural numbers (written $\text{age } : G \rightarrow \mathbb{N}$), since every member of the group is in the domain of the function.

A function (partial or total) is one-to-one (written $\mapsto$ or $\rightarrow$, respectively) if each range element is related to at most one domain element. Only if a relation $R$ is a one-to-one function is the inverse relation $R^\sim$ also a function. Assuming the situation at a given time is considered, the wife-husband relation is a one-to-one (partial) function between
A \rightarrow B \quad \text{partial function}
A \rightarrow B \quad \text{total function}
a 

\quad \text{partial one-to-one function}
A \leftrightarrow B \quad \text{total one-to-one function}
A \Rightarrow B \quad \text{partial onto function}
A \Rightarrow B \quad \text{total onto function}

Table 4.1: Function symbols

groups of people. To complete the picture, a function is \textit{onto} if it covers the entire target as its range. Thus, if \( f : X \rightarrow Y \) and \( \text{ran} f = Y \), the function \( f \) is onto \( Y \), written \( f : X \Rightarrow Y \), and analogously for total onto functions. For any finite group of people \( G \), the \textsl{child-parent} function from \( G \) to \( G \) can \textit{not} be onto \( G \) (there must be a person in \( G \) who is not the parent of anyone else in \( G \), or else there would have to be a cycle with someone an ancestor of himself).

Since a function is simply a special case of a relation, which in turn is a special kind of set, all set or relation operations can be applied to functions. In general, however, the result will not be a function. For example

\[{(0, 5), (2, 6), (4, 5)} \cup {(1, 5), (2, 4), (3, 4)}\]

is not a function, even though both arguments are, because the domain element 2 is mapped to both 6 and 4. Some special notation is introduced for functions, to express common operations more concisely. First, the common application of a function \( f \) to an argument \( a \), written \( f a \), is a concise way to express the range element of \( f \) that corresponds to the domain element \( a \). The more common form that puts the argument in parentheses is only used in Z when necessary to avoid ambiguity.

A common operation on functions that arises in many specifications is known as “functional override”, written as \( f \oplus g \). Again viewing the functions as sets of pairs, this is formally defined as

\[f \oplus g = ((\text{dom } g) \triangle f) \cup g.\]
This is simply $g$ combined with the part of $f$ not in conflict with $g$. That is, the pairs of $f$ that would cause the union to not be a function are not included in the result. Note that this operation is not symmetric and that the second function argument ‘overrides’ the conflicting part of the first function.

Unsurprisingly, sequences of elements in $\mathbb{Z}$ are also ultimately viewed as particular kinds of sets. More specifically, a sequence is a partial function from the natural numbers to the elements of the sequence, where the range is exactly $1 \ldots n$ if there are $n$ elements in the sequence (and the range is called the index of the sequence). Thus, a sequence of letters $AXBYAB$ would be represented by the set

$$\{ 1 \mapsto A, 2 \mapsto X, 3 \mapsto B, 4 \mapsto Y, 5 \mapsto A, 6 \mapsto B \}$$

The set of all possible sequences with elements from the set $S$ is defined by

$$\text{seq}(S) \doteq \{ f : \mathbb{N} \to S \mid \text{dom } f = 1 \ldots \#f \}$$

Clearly, any operations on sets, relations, or functions can also be performed on sequences, but the result will not necessarily be a sequence. In addition, there are a few special operations that are usual for sequences, of which we shall need bracketing and concatenation.

Brackets around an element or list of elements define a sequence with those elements in their order of appearance, as in

$$\langle x, y \rangle \doteq \{ 1 \mapsto x, 2 \mapsto y \}$$

The usual concatenation operator between two sequences is denoted by

$$s \circ t \doteq s \cup \{ i : \mathbb{N} \mid i \in 1 \ldots \#t \cdot (i + \#s) \mapsto (ti) \}$$

where the elements of the sequence $t$ are after the elements of the sequence $s$. Note that the definition above is a union of the pairs that make up the sequence $s$, along with new pairs derived from those in the sequence $t$ but with each index value increased by the number of elements in $s$ ($\#s$).
Now enough notation has been introduced to allow a first reasonable specification. In Figure 4.1, a schema describing a Library is presented. It assumes the existence of three sets called Copy, Book, and Reader describing, respectively, a physical copy of a book, (e.g., a running unique identifying number), abstract information about a book (title, author, ...) and information about a reader (name, address, library card number, ...) The predicate part of this schema can be viewed as an invariant of the library. The variables define a state, and the predicate determines what must be true of this state.

What is clearly missing is a description of the operations in a library. For this purpose, Z provides a special facility known as variable augmentation, to be used only in schemas and proofs relating to operations. In this context, a variable declaration with an ‘unadorned’ variable represents the state before the operation, while a version with a ’ (called a primed version) relates to the variable after the operation (just as in the Larch interface languages). In addition, any variable with a question mark (?) after it is intended to be an external input for the operation, and a variable with an exclamation point (!) is an output for the operation. Note the distinction between the (internal) primed state after the operation, and (external) output. This distinction is often useful, but occasionally burdensome, when it is not clear whether the operation will be embedded within another more complex
one, so that the apparently external result becomes internal.

Now a schema describing an operation, such as borrowing a book, can be defined. All variables describing the library need to be defined both in the state before the operation, and in a primed version, for the state afterwards. Clearly, all invariants of the library are required to hold both among the regular variables, and among the primed versions (i.e., in the states before and after the operation, respectively). Input for such an operation would be a particular copy of a book, say $c?$, and a variable denoting a reader, say, $r?$. Conditions that should hold in order for the operation to occur might include that the copy of the book is on the shelves ($c? \in \text{shelved}$), that the reader is registered at the library ($r? \in \text{readers}$), and that the reader has taken out fewer than the maximum allowed number of books ($\#(\text{issued} \ni \{r?\}) < \text{maxloans}$).

The desired effect of the operation can be specified by the requirement $\text{issued}' = \text{issued} \oplus \{c? \mapsto r?\}$, so that the copy will be associated with the borrowing reader in the new version of the function $\text{issued}$.

Rather than write out all the declarations, and both the unprimed and the primed versions of the invariants, it is possible to apply a prime to a schema. This is a shorthand for a version with a prime added to every declared variable, both in the declarations and in the predicate part. Moreover, a schema may be included within another schema—which means to include all of its declarations and its predicate. The borrow operation then becomes

\[
\begin{array}{l}
\text{Borrow} \\
\text{Library} \\
\text{Library}' \\
c? : \text{Copy} \\
r? : \text{Reader} \\
\end{array}
\]

\[
c? \in \text{shelved} ; r? \in \text{readers} ; \#(\text{issued} \ni \{r?\}) < \text{maxloans} \\
\text{issued}' = \text{issued} \oplus \{c? \mapsto r?\}
\]

As additional notation intended to save effort, we have $\Delta A$ as a shorthand for $A$ and $A'$ (so that we could have written $\Delta \text{Library}$ instead of the first two lines of the schema above).
CHAPTER 4. Z AND SET-BASED SPECIFICATION

The notation $\Xi A$ is equivalent to

$$[\Delta A \mid \text{var}A = \text{var}A'].$$

The symbol $\Xi$ represents the unprimed and primed versions of its argument, plus the assertion that the state is not changed by the operation containing such a notation. This is used for identifying the operations that examine the state, analogously to Larch. A schema that could be used in continuing the Library specification is

$$\begin{align*}
books \leftarrow \text{out} \\
\Xi \text{Library} \\
r? : \text{Reader} \\
books! : \text{F Copy} \\
r? \in \text{readers} \\
books! = \text{dom}(\text{issued} \uparrow \{r?\})
\end{align*}$$

4.4 The schema calculus

The combinations of schemas just seen in specifying the Library operations are simple examples of what is known as the schema calculus. These are rules for deriving one schema from another, and particularly for combining schemas.

*Inclusion* is the most common method of combining schemas. When the name of one schema is written in the declaration part of another, the declarations are to be merged, and a conjunction is to be taken between the predicate of the included schema, and that given explicitly in the including one. In the merge, variables that appear in both (and have the same type declaration) appear once, as do declarations that appear in either. If there is a conflict in the declarations of the same variable name, the inclusion is illegal. Thus, given a schema

$$\begin{align*}
S \\
x : \mathbb{N} \\
y : \mathbb{N} \\
x \leq y
\end{align*}$$
we may write

\[
\begin{array}{c}
T \\
\underline{S} \\
\underline{z : N} \\
\underline{z \leq x}
\end{array}
\]

This is entirely equivalent to the expanded version, namely:

\[
\begin{array}{c}
T \\
x : N \\
y : N \\
z : N \\
\underline{x \leq y \land z \leq x}
\end{array}
\]

Inclusion is used to build up schemas in stages, to encourage modularity, as well as to conveniently separate the static and the dynamic parts of a system, as seen in the static \textit{Library} schema, and its use in describing the \textit{borrow} operation.

Logical operators among schemas are also possible. For example,

\( S \land T \) is a symmetric version of inclusion, and has the same meaning.

\( S \lor T \) merges the declarations as above, but has a disjunction between the predicates

\( \neg S \) is like \( S \), but with a predicate that is the negation of \( S \)’s

A few notations to derive new schemas from old are a little less standard. Among these we have:

\( S[\text{new}/\text{old}] \) defines substitution. The result is a schema like \( S \), but with the name \textit{new} in place of free occurrences of the name \textit{old}.

\( S \setminus \{v_1, ..., v_n\} \) is called hiding. A schema is defined that is like \( S \) except that the variables \( v_1 \) through \( v_n \) are removed from the declaration, and the phrase \( \exists v_1...v_n \) precedes the predicate of \( S \).
After a hiding transformation, the hidden names no longer refer to variables from the state of the system being specified. The assertion is now merely that some values exist that will make the predicate true, with no connection to any role these names might have in defining the system. Thus for the schema \( T \) defined above, \( T \setminus \{ x \} \) is the schema

\[
\begin{align*}
z & : \mathbb{N} \\
y & : \mathbb{N}
\end{align*}
\]

\( \exists x : \mathbb{N} \cdot x \leq y \land z \leq x \)

The assertion now means that there is some natural number between \( z \) and \( y \), and is equivalent to \( z \leq y \), since \( y \) and \( z \) are already natural numbers.

For schemas that describe operations, two more derived schemas can be defined, using the notation already introduced. For a schema \( S \) describing an operation, \( \text{pre } S \) is another schema, whose predicate is the precondition for the operation specified by \( S \). It is formally defined as \( S \) hiding all declared variables with a \( ! \) or a \( ' \). Recall that these are the variables intended to represent output, and the state after the operation, respectively. By removing them from the declaration, and asserting that there exist values with those names before the predicate of \( S \), only the conditions that relate to the state before the operation are left in \( \text{pre } S \).

For the \textit{Borrow} operation defined previously, \( \text{pre } \textit{Borrow} \) is

\[
\begin{align*}
\text{Library} \\
c? : \text{Copy} \\
r? : \text{Reader}
\end{align*}
\]

\[
\exists \text{issued'} \cdot c? \in \text{shelved} \land r? \in \text{readers} \land \\
\#(\text{issued} \triangleright \{r?\}) < \text{maxloans} \land \\
\text{issued'} = \text{issued} \oplus \{ c? \mapsto r? \}
\]

Note that the primed version of \textit{Library} has been removed, and that again the name \( \text{issued'} \) is quantified in the predicate, and has
nothing to do with the part of the system state \textit{issued}. In fact, the
existential quantifier and the second part of the predicate merely assert
that \textit{issued} $\oplus \{ e? \rightarrow r? \}$ is a function. This is trivially true from
the definition of $\oplus$ and the fact that the arguments are functions, and thus
can be removed, leaving the much simpler

\begin{verbatim}
| Library |
c? : Copy |
r? : Reader |
c? $\in$ shelved ; r? $\in$ readers ; #(issued $\triangleright \{r?\}) < maxloans |
\end{verbatim}

This is a special case of applying the reduction

$$\exists x : S \bullet (x = T \land P) \Leftrightarrow T \in S \land P[T/x]$$

That is, in the assertion $P$, we may replace the existentially quantified
variable $x$ by the expression $T$ to which it is equal, along with the type
information, if necessary, and eliminate the quantification.

The final combination of schemas we shall need is known as \textit{schema
composition}. Recall that for relations, their composition $R \circ S$ is simply
the transitive pairs $(x, z)$ for which there is a $y$ satisfying $(x, y) \in R$
and $(y, z) \in S$. The same idea applies to composition of schemas that
represent operations, except that the primed version of variables from
the first component will be associated with the unprimed versions in
the second component. For example, given schemas $A$ and $B$, each with
declarations for $x$ and $x'$, a new name can be used to identify the $x'$ of
$A$ with the unprimed $x$ of $B$. The composition then is the conjunction
of the schemas, hiding the variable just introduced. That is

$$A \circ B \equiv (A[new/x'] \land B[new/x]) \setminus \{new\}$$

The composition then is left with the $x$ of $A$, and the $x'$ of $B$, where
the former result of $A$ is connected to the former initial state of $B$ by
the hidden variable $new$. As above, often the existential quantification
can be simplified or eliminated. New names and hiding as above should
be used for all primed variables declared in the first component of the
composition which also are declared with unprimed versions in the second component. All other variables in both components are unchanged. Consider a simple schema defined by

$F$

\[
s, s', i? : \mathbb{N} \\
\hline
s' = i? + s
\]

and another defined by

$T$

\[
s, s', o! : \mathbb{N} \\
\hline
s' = 2 \times s \\
o! = s'
\]

Then the composition $F \circ T$ is

\[
s, s', i?, o! : \mathbb{N} \\
\hline
\exists \text{new} : \mathbb{N} \cdot \text{new} = i? + s \land s' = 2 \times \text{new} \land o! = s'
\]

Here the name $\text{new}$ is substituted for $s'$ of $F$, and $s$ of $T$. As previously, the predicate can be simplified to

\[
s' = 2 \times (i? + s) \land o! = s'
\]

If we define a simple schema $Register$ by

$Register$

\[
\begin{array}{c}
\text{Library} \\
\text{Library'} \\
r? : \text{Reader} \\
\end{array}
\]

\[
\text{readers'} = \text{readers} \cup \{r?\}
\]

Then the composition $Register \circ Borrow$, after simplification, is
This schema has the effect of registering a reader, and then checking out a book by that reader, without having to check that the new reader is already in $\text{readers}$.

### 4.5 Examples

#### 4.5.1 A symbol table

A symbol table can be considered as a way of associating strings of letters (possibly representing variables or labels in a high level programming language) with values (e.g., internal memory locations or labels). Thus we have

$$ST = STR \rightarrow VAL$$

The standard symbol table functions of adding an entry (called here $\text{Enter}$) and looking up a value for a given string (Lookup) can then be defined by

$$\text{Enter}$$

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$st' = st \oplus { s? \mapsto v? }$</td>
<td></td>
</tr>
</tbody>
</table>
Note that the predicate of \textit{Lookup} includes both conditions on the input and the relation between the input and the output. Recall that the application of the function \( st \) to the argument \( s? \) is indicated simply by juxtaposition, without using parentheses. An operation to initialize a symbol table could be defined as

\[
\text{Init} \equiv [\text{st'} : \text{ST} \mid \text{st'} = \{\}] 
\]

As given, the operation \textit{Lookup} is not defined when the argument \( s? \) is not in the domain of the symbol table. The schema calculus is intended to encourage a modular treatment of issues such as error checking and treatment of exceptional inputs. In this example it would be reasonable to have a schema of an operation that only treats input strings never entered in the symbol table:

Then a more robust lookup operation could be

\[
\textit{Rlookup} \equiv \textit{Lookup} \lor \textit{Badquest} 
\]

Similarly, the calculus can be used to add functionality or simply for debugging purposes by having a \textit{Log} schema.
4.5. EXAMPLES

Then we might define

\[ \text{Auglookup} \equiv (\text{Lookup} \land \text{Log}) \lor \text{Badquest} \]

### 4.5.2 A Stack

A collection of schemas for the stack operations could use a sequence to represent the stack and then define

\[ \text{push} \]

\[ x? : E ; s, s' : \text{seq}(E) \]

\[ s' = s \cup \{(\#s + 1) \mapsto x?\} \]

That is, the \text{push} operation is represented by adding the new element to the end of the sequence representing the contents of the stack. Recall from the definitions of concatenation and bracketing, that this is equivalent to \( s \triangleleft (x?) \).

\[ \text{pop} \]

\[ s, s' : \text{seq}(E) \]

\[ s' = \#s \triangleleft s \]

The \text{pop} operation removes the last element from the sequence \( s \) by using an operation on relations: the set \( s \) without the pair whose index value is \( \#s \).

\[ \text{top} \]

\[ x! : E ; s : \text{seq}(E) \]

\[ \exists s \]

\[ x! = s (\#s) \]
Note that here we choose to add, remove, and examine the stack elements from the end of the sequence. An equivalent specification could be written adding and removing elements from the beginning of the sequence. Even though the specification would be slightly more complicated to write, the 'efficiency' of a specification is irrelevant. Still, this need to commit to one end or the other seems less abstract than the Larch version: one or the other possibility has to be used, and this may 'prejudice' which kind of implementation is chosen. Changing this to a specification of a queue is a matter of adding to one end of the sequence and removing from the other. It is also simple to specify in this way more complex structures such as double-ended queues.

### 4.5.3 Elements of an assembler specification

Another example of the view of sequences as special kinds of functions, which are also of course relations, can be seen in an assembly language program. We are given that the program is a sequence of commands, and that commands have a partial projection function $\textit{label}$ that returns the symbolic label of any command that has a label, and otherwise is undefined. If the input program is denoted by $\textit{inprog}$, we can define a specialized symbol table of labels differently than the general table seen previously where elements can be added one by one. Here we can define the contents of the table as a function of the entire program, viewed as a sequence:

$$\textit{symtab} \triangleq (\textit{inprog} \circ \textit{label})^*$$

Since the program is a sequence of commands, it might have the form

$$\{1 \mapsto (L_1 : \textit{load } x), 2 \mapsto (\textit{add } y), 3 \mapsto (L_2 : \textit{store } z), 4 \mapsto (\textit{goto } L_5)\ldots\}$$

The composition of $\textit{inprog}$ and $\textit{label}$ would give the function (no longer a sequence):

$$\{1 \mapsto L_1, 3 \mapsto L_2, \ldots\}$$

and the inverse would give a relation from labels to sequence (line) numbers, as required for the labels in a symbol table. If the labels are unique, this also would be a function, so that $\textit{symtab} L_2$ returns 3.
If there is also a projection function \( \text{refer} \) that returns the label that appears in the argument field of an assembly language command (e.g., the \( L5 \) in the fourth command above), then one part of the predicate in the schema \( \text{ASSEMBLER} \) could be:

\[
\text{ran}(\text{inprog} \circ \text{refer}) \subseteq \text{dom symtab}
\]

This is part of the precondition of the assembler, and requires that all references to labels that appear in the input assembler program are actually labels of some statement. Assuming that the output machine language program is produced in the variable \( \text{outmach}! \), that the translation is syntactically line for line (so that the same line numbers apply), and that there is an \( \text{operand} \) projection function for the operands of machine language instructions, another part of the schema could be

\[
(\text{inprog} \circ \text{refer} \circ \text{syntab}) \subseteq (\text{outmach}! \circ \text{operand})
\]

The left side of the inclusion gives the pairs of line numbers of an instruction and a translation of a reference to a line number using the symbol table, while the right side also gives pairs of line numbers, from the machine language lines, and the corresponding operand field. This expresses the requirement from the argument field for all instructions that have abstract references. Along with added predicates that cover the other kinds of instructions, (e.g., those with constants in the argument field) this will be the specification of the assembler.

### 4.6 Refinement

Most of the specification efforts undertaken in \( Z \) have involved analyzing informal requirements and using them as a basis to write a collection of schemas. However, there is also an associated theory of refinement for the notation. The idea is to develop a system by starting with high level \( Z \) schemes describing abstract operations, and then use the theory of refinement to replace those by schemas closer to an implementation. The theory provides precise criteria for checking that the refinement satisfies the key properties of the more abstract version.

The first question to be considered is what exactly is expected of a refinement. An interesting distinction is made in this theory between
data refinement and operation refinement. The former corresponds to
the replacement of data structures from an upper, abstract level, by
those from a lower, concrete one, and will be treated using mapping
functions analogously to the approach seen in Larch.

To understand the idea of operation refinement, we begin by defining
relation refinement. For two operations $R_1$ and $R_2$, defined over the
same domain and range, $X \leftrightarrow Y$, $R_1$ is refined by $R_2$ (written $R_1 \subseteq R_2$) if and only if:

**(Applicability)** $\text{dom } R_1 \subseteq \text{dom } R_2$

(whenever $R_1$ can be applied, so can $R_2$)

**(Correctness)** $((\text{dom } R_1) \times R_2) \subseteq R_1$

(when $R_1$ can be applied, but $R_2$ is instead, the result is in the
relation $R_1$)

The definition is reasonable when the relations are to be used to connect
domain and range elements. The definition above means that $R_2$ can
always be substituted for $R_1$ (since its domain is defined whenever $R_1$’s
is), and will produce a subset of the results possible under $R_1$. Note
that $R_2$ can have fewer possible range elements that correspond to a
given domain element also in the domain of $R_1$, but it must have at
least one. This becomes clearer when the same ideas are applied to
operations on the same state.

If $Op_1$ and $Op_2$ are operations defined by schemas, then $Op_1$ is
refined by $Op_2$ (again, $Op_1 \subseteq Op_2$) iff

**(Applicability)** $\text{pre } Op_1 \vdash \text{pre } Op_2$

(whenever $Op_1$ can be applied, so can $Op_2$)

**(Correctness)** $\text{pre } Op_1 \land Op_2 \vdash Op_1$

(when $Op_1$ can be applied, but $Op_2$ is instead, the result could
have been obtained by applying $Op_1$)

The requirements are the same as for relations, but using the notation
of logic and schemas instead of that for sets. There can be less
nondeterminism (fewer possible results) in the refinement because it
is more specific or concrete. However, a refinement cannot ‘refuse to
implement a legal input of the more abstract level, and must produce some result.

Consider a schema Takesome defined by

\[
\begin{array}{c}
\text{Takesome} \\
\hline
x, x' : \mathbb{N} \\
0 < x' < x
\end{array}
\]

This schema is satisfied by any value of \( x' \) greater than zero and strictly less than \( x \). Its precondition would hide \( x' \) but assert that there must be some integer value between zero and \( x \), and thus requires that \( x \) is at least two.

A possible operation refinement could be the schema Takeone defined by

\[
\begin{array}{c}
\text{Takeone} \\
\hline
x, x' : \mathbb{N} \\
x' = 1
\end{array}
\]

Note that Takeone is defined for values of \( x \) for which Takesome is not (namely zero and one). However, when both are defined, the value of \( x' \) for Takeone (namely, one) is a possible value if Takesome had been applied instead.

This may seem quite theoretical, but consider specifying a scheduler for jobs in an operating system, where many orderings are possible, but some basic responsiveness and fairness properties are required. In practice, a specific round-robin scheduler may be applied, which satisfies the required properties, but has fewer possible orderings of the jobs than indicated by the abstract requirements. It is not difficult to see that the notion of refinement seen here is appropriate for such multiprocess systems.

Still, it is insufficient to only consider operation refinements. There also are data refinements where an abstract state representation is replaced by a more concrete version. For example, on the abstract level we might consider a set of elements, and on a more concrete level implement the set using a sequence or array. An abstract state representation could have the schema:
A more concrete state representation might have a schema that uses a domain \( \text{STATUS} \) defined by

\[
\text{STATUS} \doteq \{ \text{reg}, \text{ex} \}
\]

The schema then might be:

\[
\begin{array}{l}
\text{Concgroup} \\
\text{memseq} : \text{seq} \cdot \text{NAMES} \\
\text{stat} : \text{NAMES} \rightarrow \text{STATUS} \\
\#\{ y : \text{NAMES} \mid y \in \text{ran} \text{memseq} \wedge \text{stat} y = \text{ex} \} \leq \text{limit}
\end{array}
\]

To connect the two states, a mapping function, traditionally from the concrete representation to the abstract one as in Larch, is needed. This is given as yet another schema:

\[
\begin{array}{l}
\text{Connect} \\
\text{Absgroup} \\
\text{Concgroup} \\
\text{members} = \text{ran} \text{memseq} \\
\text{exec} = \{ y : \text{NAMES} \mid y \in \text{ran} \text{memseq} \wedge \text{stat} y = \text{ex} \}
\end{array}
\]

At this point the consistency of \( \text{Connect} \) (that its predicates do not allow proving \( \text{false} \)) shows that this mapping is reasonable.

When operations are added, we use a combination of data and operation refinement. If we assume an abstract state \( \text{AS} \) and a concrete state \( \text{CS} \), each with an initialization operation \( \text{initAS} \) and \( \text{initCS} \), respectively, and other pairs of operations \( \text{Aop} \) and \( \text{Cop} \), plus a functional mapping between the states in a schema \( \text{Connect} \), the requirements for a correct refinement are to show:
for the initial states

\[ \text{initCS} \land \text{Connect} \vdash \text{initAS} \]

for each pair \( Aop \) and \( Cop \)

\[ \text{pre Aop} \land \text{Connect} \vdash \text{pre Cop} \]

\[ \text{pre Aop} \land \text{Connect} \land \text{Cop} \land \text{Connect}' \vdash \text{Aop} \]

Returning to the example, on the abstract level, a possible initialization would be

\[
\begin{array}{l}
\text{initabs} \\
\text{Absgroup'} \\
\text{members'} = \{\} \\
\text{exec'} = \{\}
\end{array}
\]

and a typical operation is:

\[
\begin{array}{l}
\text{Abselect} \\
\Delta \text{Absgroup} \\
x? : \text{NAMES} \\
x? \in \text{members} \\
\#(\text{exec} \cup \{x?\}) \leq \text{limit} \\
\text{exec'} = \text{exec} \cup \{x?\} \\
\text{members'} = \text{members}
\end{array}
\]

The corresponding schemas on the concrete level could be:

\[
\begin{array}{l}
\text{initcon} \\
\text{Congroup'} \\
\text{memseq'} = \{\} \\
\text{stat'} = \{\}
\end{array}
\]
and an operation:

\[
\begin{align*}
\text{Concelect} \\
\Delta \text{Congroup} \\
x? : \text{NAMES} \\
x? \in \text{ran memseq} \\
(\text{stat } x? = \text{ex}) \lor \#\{y : \text{NAMES} \mid y \in \text{ran memseq} \land \text{stat } y = \text{ex}\} < \text{limit} \\
\text{stat}' = \text{stat} + \{x? \mapsto \text{ex}\}
\end{align*}
\]

It is left to the reader to prove that these constitute a correct refinement according to the criteria above.

The refinement calculus stays within the Z formalism. Little attention has been paid to the question of whether a C implementation satisfies a Z specification. However, it is possible to adopt the Larch division to a core specification notation along with interface languages that use the notation to augment e.g., input/output specifications of C modules. This possibility will be more closely examined in a later chapter.