CHAPTER 10

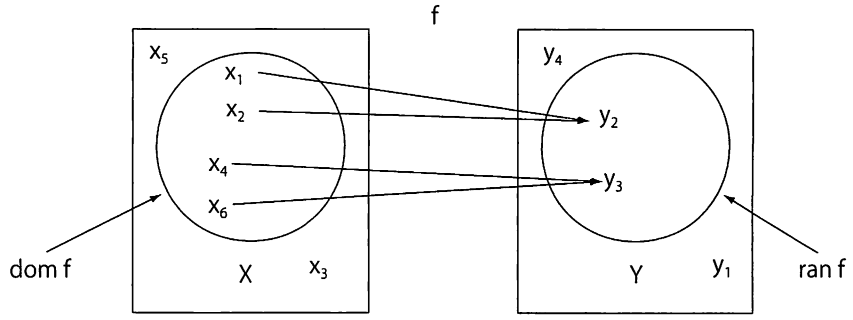
Functions

10.1 A function is a relation

In a programming language a *function* is a way of specifying some processing which produces a value as a result. In Z a function is a data structure. These two views are not incompatible; the programming language view is just a restricted form of the Z view and in both cases a function provides a result value, given an input value or values.

A function is a special case of a relation in which there is *at most one* value in the range for each value in the domain. A function with a finite domain is also known as a *mapping*.

Figure 10.1



Note that in the diagram the lines do not diverge from left to right.

In Z a function *f* from the type *X* to the type *Y* is declared by:

$$f: X \rightarrow\!\!\!\!\!+ \; Y$$

and is pronounced 'the function *f*, from *X* to *Y*'. This is equivalent to the relation *f* from *X* to *Y*:

$$f : X \leftrightarrow Y$$

with the restriction that for each *x* in the domain of *f*, *f* relates *x* to *at most one y*:

$$x \in \text{dom } f \Rightarrow \exists_1\, y: Y \bullet x\, f\, y$$

10.1.1 Examples of functions

The relation between persons and identity numbers:

$$identityNo: PERSON \leftrightarrow \mathbb{N}$$

is a function if there is a rule that a person may only have one identity number. In this case it would be declared:

identityNo: PERSON $\rightarrowtriangle$ $\mathbb{N}$

There would probably also be a rule that only one identity number may be associated with any person, but this is not indicated here.

Note that there is no restriction that the function must map different values of the source on to different values of the target. So one could have the function giving a person's mother:

hasMother: PERSON $\rightarrowtriangle$ PERSON

since any person can have only one mother, but several people may have the same mother.

10.2 Function application

All the concepts which pertain to relations are also applicable to functions. In addition, however, a function may be *applied*. Since there will be at most one value in the range for a given *x* it is possible to designate that value directly. The value of *f* applied to *x* is the value in the range of the function *f* corresponding to the value *x* in its domain. The application is undefined if the value of *x* is not in the domain of *f*. It is important to check that a value *x* is in the domain of the function *f* before attempting to apply *f* to *x*.

The application of the function *f* to the value *x* (called its *argument*) is written:

f x

and pronounced '*f* of *x*' or '*f* applied to *x*'.

Some people put brackets around the argument, as in:

f(x)

but although this is allowed, it is not necessary.

We check the applicability of *f* by writing in this style:

$x \in \text{dom } f$
$f\,x = y$

10.3 Partial and total functions

The functions given as examples above have been shown as *partial*, which means that there may be values of the source which are not in the domain of the function.

A *total* function is one where there is a value for every possible value of *x*, so *f x* is always defined. The domain is the whole of the source. It is declared by

$$f: X \rightarrow Y$$

and is pronounced '*f* is a total function from *X* to *Y*'. It is equivalent to the partial function from *X* to *Y* with the restriction that

$$\mathrm{dom}\ f = X$$

Because of this, function application can *always* be used with a total function.

10.3.1 Examples of total functions

The function *age*, from *PERSON* to natural numbers, is total since every person has an age:

$$\mathrm{age}: \mathrm{PERSON} \rightarrow \mathbb{N}$$

The function *hasMother*, given above, is total since for any person there is exactly one person who is or was that person's mother:

$$\mathrm{hasMother}: \mathrm{PERSON} \rightarrow \mathrm{PERSON}$$

10.4 Other classes of functions

In addition to total or partial, other classes of functions can be distinguished, as discussed below.

10.4.1 Injection

An *injection*, or *injective function*, is a function which maps different values of the source on to different values of the target.

$$f: X ⤔ Y$$

The inverse relation of an injective function *f*, from *X* to *Y*, *f*$^\sim$, is itself a function, from *Y* to *X*:

$$f^\sim \in Y ⇸ X$$

An injective function may be partial:

$$f: X ⤔ Y$$

or total:

$$f: X \rightarrowtail Y$$

Injective functions are sometimes described as 'one-to-one'.

Since it is likely that each number is associated with only one person, the function *identityNo* given above would be injective. Monogamous marriage can be expressed as an injective function relating husband to wife (or vice versa).

10.4.2 Surjection

A *surjection*, or *surjective function*, is a function for which its range is the whole of its target. Given the surjective function *f* from *X* to *Y*

$\text{ran } f = Y$

A surjective function may be partial:

$f: X ⤀ Y$

or total:

$f: X \twoheadrightarrow Y$

Example Given

SIDE ::= left | right

the function

driveOn: COUNTRY ⤀ SIDE   side of the road one drives on

is a *surjection*, because the range includes *all* the values of the type *SIDE*. (There are some countries where one drives on the left and some where one drives on the right). You might consider that it is also *total*, since there are no countries for which the side of the road is not defined:

driveOn: COUNTRY ↠ SIDE

10.4.3 Bijection

A *bijection*, or *bijective function*, is one which maps every element of the source on to every element of the target in a one-to-one relationship. It is therefore: injective, total and surjective:

$f: X ⤖ Y$

An example of a bijection is a foreign-language vocabulary test that involves finding the pairings between a word in one language and its counterpart in another, where there are to be no left-over words in either list.

10.5 Constant functions

Some functions are used as a means of providing a value, given a parameter or parameters. These are usually functions that maintain a *constant* mapping from their input parameters to their output values. If the value of the mapping is known, a value can be given to the function by an *axiomatic definition*. For example, to define the function *square*:

$$
\begin{array}{|l}
square: \mathbb{Z} \rightarrow \mathbb{N} \\
\hline
\forall n: \mathbb{Z} \bullet \\
\quad square\ n = n * n
\end{array}
$$

Where convenient, several functions can be combined in one definition:

$$
\begin{array}{|l}
square: \mathbb{Z} \rightarrow \mathbb{N} \\
cube: \mathbb{Z} \rightarrow \mathbb{Z} \\
\hline
\forall n: \mathbb{Z} \bullet \\
(square\ n = n * n \wedge \\
\quad cube\ n = n * n * n)
\end{array}
$$

10.6 Overriding

A function or relation can be modified by adding mapping pairs to it or by removing pairs. It can also be modified so that for a particular set of values of the domain it has new values in the range. This is called *overriding*.

For functions or relations *f* and *g*, both of the same type, *f* overridden by *g* is written:

$$f \oplus g$$

It is the same as *f* for all values that are not in the domain of *g*, and the same as *g* for all values that are in the domain of *g*:

If

$$x \in \mathrm{dom}\ f \wedge x \notin \mathrm{dom}\ g$$

then

$$f \oplus g\ x = f\ x$$

But if

$$x \in \mathrm{dom}\ g$$

then

$$f \oplus g\ x = g\ x$$

A definition is:

$$f \oplus g = (\mathrm{dom}\ g \lhd f) \cup g$$

Note that if *f* and *g* have disjoint domains

$$\text{dom } f \cap \text{dom } g = \emptyset$$

then

$$f \oplus g = f \cup g$$

Example The recorded age of person *p?* is increased by 1:

$$\text{age} \oplus \{p? \mapsto \text{age } p? + 1\}$$

The function *age* is overridden by the function with only *p?* in its domain which maps *p?* to its former value plus 1.

10.7 Example from business – stock control

A warehouse holds stocks of various items *carried* by an organisation. A computer system records the *level* of all items carried, the *withdrawal* of items from stock and the *delivery* of stock. Occasionally, a new item will be carried and items will be discontinued, provided that their stock level is zero.

[ITEM] the set of all items (item codes)

$$\begin{array}{|l}
\textbf{Warehouse} \\
\hline
\text{carried}: \mathbb{P}\,\text{ITEM} \\
\text{level}: \quad \text{ITEM} \twoheadrightarrow \mathbb{N} \\
\hline
\text{dom level} = \text{carried} \\
\hline
\end{array}$$

Every item carried has a level, even if it is zero.

$$\begin{array}{|l}
\textbf{Init} \\
\hline
\text{Warehouse}' \\
\hline
\text{carried}' = \emptyset \\
\text{level}' = \emptyset \\
\hline
\end{array}$$

Initially there are no items.

$$\begin{array}{|l}
\textbf{Withdraw} \\
\hline
\Delta\text{Warehouse} \\
i?: \quad \text{ITEM} \\
qty?: \quad \mathbb{N}_1 \\
\hline
i? \in \text{carried} \\
\text{level } i? \geq qty? \\
\text{level}' = \text{level} \oplus \{i? \mapsto (\text{level } i? - qty?)\} \\
\text{carried}' = \text{carried} \\
\hline
\end{array}$$

For a quantity of an item to be withdrawn, the item must be carried and there must be enough stock.

Deliver

ΔWarehouse
$i?: \quad ITEM$
$qty?: \quad \mathbb{N}_1$

$i? \in carried$
$level' = level \oplus \{i? \mapsto (level\ i? + qty?)\}$
$carried' = carried$

Only deliveries for carried items are accepted. There is no upper limit on stock held.

CarryNewItem

ΔWarehouse
$i?: \quad ITEM$

$i? \notin carried$
$level' = level \cup \{i? \mapsto 0\}$
$carried' = carried \cup \{i?\}$

A new item must not already be carried and will initially have a level of zero.

DiscontinueItem

ΔWarehouse
$i?: \quad ITEM$

$i? \in carried$
$level\ i? = 0$
$carried' = carried \setminus \{i?\}$
$level' = \{i?\} \lhd level$

An item to be discontinued must currently be carried and must have a level of zero.
Note: errors have not been handled in this simple version.

10.8 Example from data processing

10.8.1 Indexed-sequential files

In the programming language COBOL a type of data file called an indexed-sequential file is available. In principle an indexed-sequential file is a sequence of records which can be accessed in any order by specifying the value of a field of the desired record, called the *key*. There is at most one record in the file for any value of the key.

Operations are available to read, write, insert and delete records (using the COBOL instructions *READ*, *REWRITE*, *WRITE* and *DELETE* respectively). The behaviour of these operations can be specified as follows:

[KEY]	set of all keys for this file
[DATA]	remaining fields of record (other than key)

```
__ISFile________________
| file:      KEY ⇸ DATA
|________________________
```

The file is regarded as a function from a *key* to the rest of the *data* in the record. The function is partial since there may be values of the key for which there is no record on the file.

10.8.2 Read operation

The operation *Read* is a function application:

```
__Read__________________
| ΞISFile
| k?:        KEY
| result!:   DATA
|------------------------
| k? ∈ dom file
| result! = file k?
|________________________
```

The value of *result!* is the data of the record with the key *k?*, if there is one. The first line of the predicate is to check the applicability of the function *file* to the given key. The file is unchanged.

10.8.3 Rewrite operation

The operation *Rewrite* is a functional overriding:

```
__Rewrite_______________
| ΔISFile
| k?:        KEY
| new?:      DATA
|------------------------
| k? ∈ dom file
| file' = file ⊕ {k? ↦ new?}
|________________________
```

The function is changed only for the value of *k?* in the domain, which now maps to the new data *new?*.

10.8.4 Write operation

The operation *Write* is a functional (relational) union:

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\text{Write}} \\
\hline
\Delta \text{ISFile} & \\
\text{k?:} & \text{KEY} \\
\text{new?:} & \text{DATA} \\
\hline
\multicolumn{2}{|l}{\text{k?} \notin \text{dom file}} \\
\multicolumn{2}{|l}{\text{file}' = \text{file} \cup \{\text{k?} \mapsto \text{new?}\}} \\
\hline
\end{array}
$$

10.8.5 Delete operation

The operation *Delete* is a functional (relational) domain anti-restriction:

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\text{Delete}} \\
\hline
\Delta \text{ISFile} & \\
\text{k?:} & \text{KEY} \\
\hline
\multicolumn{2}{|l}{\text{k?} \in \text{dom file}} \\
\multicolumn{2}{|l}{\text{file}' = \{\text{k?}\} \lhd \text{file}} \\
\hline
\end{array}
$$

10.8.6 Error conditions

The handling of errors has not been included here, to keep the specification simple. The schemas could easily be extended to include a report of success or failure.

10.8.7 Further facilities

The COBOL language allows further operations on indexed-sequential files which make use of the fact that the records of such a file are held in order (ordered on the key). Clearly, the specification used above would not suffice to specify those operations. However, it gives a very concise explanation of the simple operations.

10.9 Summary of notation

$X \nrightarrow Y$	the set of partial functions from X to Y: $== \{ f: X \leftrightarrow Y \mid (\forall x: X \mid x \in \operatorname{dom} f \bullet (\exists_1 y: Y \bullet x \, f \, y)) \bullet f \}$
$X \rightarrow Y$	the set of total functions from X to Y: $== \{ f: X \nrightarrow Y \mid \operatorname{dom} f = X \bullet f \}$
$X ⤔ Y$	the set of partial injective functions from X to Y: $== \{ f: X \nrightarrow Y \mid f^{\sim} \in Y \nrightarrow X \bullet f \}$
$X \rightarrowtail Y$	the set of total injective functions from X to Y: $== \{ f: X ⤔ Y \mid \operatorname{dom} f = X \bullet f \}$
$X ⤀ Y$	the partial surjective functions from X to Y: $== \{ f: X \nrightarrow Y \mid \operatorname{ran} f = Y \}$
$X \twoheadrightarrow Y$	the total surjective functions from X to Y: $== \{ f: X ⤀ Y \mid \operatorname{dom} f = X \}$
$X ⤖ Y$	the bijective functions from X to Y (total, injective and surjective) $== (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$
$f\,x$ *or* $f(x)$	the function f applied to x
$f \oplus g$	functional overriding $== (\operatorname{dom} g \lhd f) \cup g$

EXERCISES

1. A system records the bookings of hotel rooms on one night.
   Given the basic types:

[ROOM]	the set of all the rooms in the hotel
[PERSON]	the set of all possible persons

   the state of the hotel's bookings can be represented by the following schema:

```
__Hotel______________________
| bookedTo:      ROOM ⇸ PERSON
|____________________________
```

   (a) Explain why *bookedTo* is a *function*.
   (b) Explain why the function is *partial*.

2. An initialisation operation is:

```
__Init__________________
| Hotel'
|_______________________
| bookedTo' = ∅
|_______________________
```

and a first version of the operation to accept a booking is:

```
__AcceptBooking₀__________________
| ΔHotel
| p?:      PERSON
| r?:      ROOM
|_________________________________
| r? ∉ dom bookedTo
| bookedTo' = bookedTo ∪ {r? ↦ p?}
|_________________________________
```

Explain the meaning and purpose of each line of the schema above.

3. Write a schema *CancelBooking*$_0$ which cancels a booking made for a given person and a given room. It should deal with error conditions in the same manner as *AcceptBooking*$_0$.
4. Explain the meaning and purpose of each line of your schema *CancelBooking*$_0$.
5. The following partially specifies the Olympic games in held Sydney in the year 2000:

[COUNTRY]	the set of all countries of the world
[PERSON]	the set of all uniquely identified persons
[EVENT]	the set of all sporting events

```
__Sydney2000______________________________
| participating:   ℙCOUNTRY
| events:          ℙEVENT
| represents:      PERSON ↔ COUNTRY
| competesIn:      PERSON ↔ EVENT
| won:             EVENT ↔ PERSON
|_________________________________________
| ???
|_________________________________________
```

A person can only represent one participating country. A person can only compete in an event if he or she is representing a country and if the event is one of those of the Sydney 2000 games. Only one of the competitors in an event can win it.

(a) All the relationships have been given as general relations. Consider which of these might be better specified as functions. Re-write the signature of the schema *Sydney2000* showing this and justify your changes.
(b) Consider the relationships between the variables of your schema *Sydney2000* and add predicates to represent these. Justify your predicates.
(c) Write a schema *JoinGames* which records a new country joining in the Sydney games.
(d) Write a schema *Win* that records the winner of an event.
(e) Write a schema *CountryGolds* that supplies the number of events that have been won by representatives of a given country.