1.2 The birthday book

The best way to see how these ideas work out is to look at a small example. For a first example, it is important to choose something simple, and I have chosen a system so simple that it is usually implemented with a notebook and pencil rather than a computer. It is a system which records people's birthdays, and is able to issue a reminder when the day comes round.

In our account of the system, we shall need to deal with people's names and with dates. For present purposes, it will not matter what form these names and dates take, so we introduce the set of all names and the set of all dates as *basic types* of the specification:

$$[NAME, DATE].$$

This allows us to name the sets without saying what kind of objects they contain. The first aspect of the system to describe is its *state space*, and we do this with a schema:

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

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

Like most schemas, this consists of a part above the central dividing line, in which some variables are declared, and a part below the line which gives a relationship between the values of the variables. In this case we are describing the state space of a system, and the two variables represent important *observations* which we can make of the state:

- *known* is the set of names with birthdays recorded;
- *birthday* is a function which, when applied to certain names, gives the birthdays associated with them.

The part of the schema below the line gives a relationship which is true in every state of the system and is maintained by every operation on it: in this case, it says that the set *known* is the same as the domain of the function *birthday* – the set of names to which it can be validly applied. This relationship is an *invariant* of the system.

In this example, the invariant allows the value of the variable *known* to be derived from the value of *birthday*: *known* is a *derived* component of the state, and it would be possible to specify the system without mentioning *known* at all. However, giving names to important concepts helps to make specifications more readable; because we are describing an abstract view of the state space of the birthday book, we can do this without making a commitment to represent *known* explicitly in an implementation.

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

$$known = \{ \text{John}, \text{Mike}, \text{Susan} \}$$

$$\begin{aligned} birthday = \{ &\text{John} \mapsto \text{25–Mar}, \\ &\text{Mike} \mapsto \text{20–Dec}, \\ &\text{Susan} \mapsto \text{20–Dec} \}. \end{aligned}$$

The invariant is satisfied, because *birthday* records a date for exactly the three names in *known*.

Notice that in this description of the state space of the system, we have not been forced to place a limit on the number of birthdays recorded in the birthday book, nor to say that the entries will be stored in a particular order. We have also avoided making a premature decision about the format of names and dates. On the other hand, we have concisely captured the information that each person can have only one birthday, because the variable *birthday* is a function, and that two people can share the same birthday as in our example.

So much for the state space; we can now start on some *operations* on the system. The first of these is to add a new birthday, and we describe it with a schema:

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$

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

The declaration $\Delta BirthdayBook$ alerts us to the fact that the schema is describing a *state change*: it introduces four variables *known*, *birthday*, $known'$ and $birthday'$. The first two are observations of the state before the change, and the last two are observations of the state after the change. Each pair of variables is implicitly constrained to satisfy the invariant, so it must hold both before and after the operation. Next come the declarations of the two inputs to the operation. By convention, the names of inputs end in a question mark.

The part of the schema below the line first of all gives a *pre-condition* for the success of the operation: the name to be added must not already be one of those known to the system. This is reasonable, since each person can only have one birthday. This specification does not say what happens if the pre-condition is not satisfied: we shall see later how to extend the specification to say that an error message is to be produced. If the pre-condition is satisfied, however, the second line says that the birthday function is extended to map the new name to the given date.

We expect that the set of names known to the system will be augmented with the new name:

$$known' = known \cup \{name?\}.$$

In fact we can *prove* this from the specification of *AddBirthday*, using the invariants on the state before and after the operation:

$$\begin{aligned}
known' &= \mathrm{dom}\ birthday' && \text{[invariant after]}\\
&= \mathrm{dom}(birthday \cup \{name? \mapsto date?\}) && \text{[spec. of } AddBirthday\text{]}\\
&= \mathrm{dom}\ birthday \cup \mathrm{dom}\ \{name? \mapsto date?\} && \text{[fact about 'dom']}\\
&= \mathrm{dom}\ birthday \cup \{name?\} && \text{[fact about 'dom']}\\
&= known \cup \{name?\}. && \text{[invariant before]}
\end{aligned}$$

Stating and proving properties like this one is a good way of making sure the specification is accurate; reasoning from the specification allows us to explore the behaviour of the system without going to the trouble and expense of implementing it. The two facts about 'dom' used in this proof are examples of the laws obeyed by mathematical data types:

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

$$\mathrm{dom}\{a \mapsto b\} = \{a\}.$$

Chapter 4 contains many laws like these.

Another operation might be to find the birthday of a person known to the system. Again we describe the operation with a schema:

FindBirthday

$$\begin{array}{|l}
\hline
\Xi BirthdayBook\\
name? : NAME\\
date! : DATE\\
\hline
name? \in known\\
\\
date! = birthday(name?)\\
\hline
\end{array}$$

This schema illustrates two new notations. The declaration $\Xi BirthdayBook$ indicates that this is an operation in which the state does not change: the values $known'$ and $birthday'$ of the observations after the operation are equal to their values $known$ and $birthday$ beforehand. Including $\Xi BirthdayBook$ above the line has the same effect as including $\Delta BirthdayBook$ above the line and the two equations

$$known' = known$$

$$birthday' = birthday$$

below it. The other notation is the use of a name ending in an exclamation mark for an output: the *FindBirthday* operation takes a name as input and yields the corresponding birthday as output. The pre-condition for success of the operation

is that *name*? is one of the names known to the system; if this is so, the output *date*! is the value of the birthday function at argument *name*?.

The most useful operation on the system is the one to find which people have birthdays on a given date. The operation has one input *today*?, and one output, *cards*!, which is a *set* of names: there may be zero, one, or more people with birthdays on a particular day, to whom birthday cards should be sent.

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

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

Again the Ξ convention is used to indicate that the state does not change. This time there is no pre-condition. The output *cards*! is specified to be equal to the set of all values n drawn from the set *known* such that the value of the birthday function at n is *today*?. In general, y is a member of the set $\{\, x : S \mid \ldots x \ldots \}$ exactly if y is a member of S and the condition $\ldots y \ldots$, obtained by replacing x with y, is satisfied:

$$y \in \{\, x : S \mid \ldots x \ldots \} \Leftrightarrow y \in S \land (\ldots y \ldots).$$

So, in our case,

$$m \in \{\, n : known \mid birthday(n) = today? \,\}$$
$$\Leftrightarrow m \in known \land birthday(m) = today?\,.$$

A name m is in the output set *cards*! exactly if it is known to the system and the birthday recorded for it is *today*?.

To finish the specification, we must say what state the system is in when it is first started. This is the *initial state* of the system, and it also is specified by a schema:

InitBirthdayBook

$$BirthdayBook$$

$$known = \varnothing$$

This schema describes a birthday book in which the set *known* is empty: in consequence, the function *birthday* is empty too.

What have we achieved in this specification? We have described in the same mathematical framework both the state space of our birthday-book system and the operations which can be performed on it. The data objects which appear in the system were described in terms of mathematical data types such as sets and functions. The description of the state space included an invariant relationship between the parts of the state – information which would not be part of a program implementing the system, but which is vital to understanding it.