1 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*:

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

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

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

$$\begin{aligned} &\Delta BirthdayBook \\ &name? : NAME \\ &date? : DATE \\ &\hline \\ &name? \notin known \\ &birthday' = birthday \cup \{name? \mapsto date?\} \end{aligned}$$

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 **??** 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

$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

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

$$BirthdayBook$$

$$known =$$

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.

The effects of the operations are described in terms of the relationship which must hold between the input and the output, rather than by giving a recipe to be followed. This is particularly striking in the case of the *Remind* operation, where we simply documented the conditions under which a name should appear in the output. An implementation would probably have to examine the known names one at a time, printing the ones with today's date as it found them, but this complexity has been avoided in the specification. The implementor is free to use this technique, or any other one, as he or she chooses.

2 Strengthening the specification

A correct implementation of our specification will faithfully record birthdays and display them, so long as there are no mistakes in the input. But the specification has a serious flaw: as soon as the user tries to add a birthday for someone already known to the system, or tries to find the birthday of someone not known, it says nothing about what happens next. The action of the system may be perfectly reasonable: it may simply ignore the incorrect input. On the other hand, the system may break down: it may start to display rubbish, or perhaps worst of all, it may appear to operate normally for several months, until one day it simply forgets the birthday of a rich and elderly relation.

Does this mean that we should scrap the specification and begin a new one? That would be a shame, because the specification we have describes clearly and concisely the behaviour for correct input, and modifying it to describe the handling of incorrect input could only make it obscure. Luckily there is a better solution: we can describe, separately from the first specification, the errors which might be detected and the desired responses to them, then use the operations of the Z *schema calculus* to combine the two descriptions into a stronger specification.

We shall add an extra output *result*! to each operation on the system. When an operation is successful, this output will take the value *ok*, but it may take the other values *already_known* and *not_known* when an error is detected. The following *free type definition* defines *REPORT* to be a set containing exactly these three values:

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

We can define a schema *Success* which just specifies that the result should be *ok*, without saying how the state changes:

Success

$result! : REPORT$

$result! = ok$

The conjunction operator $\wedge$ of the schema calculus allows us to combine this description with our previous description of *AddBirthday*:

$$AddBirthday \wedge Success.$$

This describes an operation which, for correct input, both acts as described by *AddBirthday* and produces the result *ok*.

For each error that might be detected in the input, we define a schema which describes the conditions under which the error occurs and specifies that the appropriate report is produced. Here is a schema which specifies that the report *already_known* should be produced when the input *name*? is already a member of *known*:

AlreadyKnown

$\Xi BirthdayBook$
$name? : NAME$
$result! : REPORT$

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

The declaration $\Xi BirthdayBook$ specifies that if the error occurs, the state of the system should not change.

We can combine this description with the previous one to give a specification for a robust version of *AddBirthday*:

$$RAddBirthday \mathrel{\widehat=} (AddBirthday \wedge Success) \vee AlreadyKnown.$$

This definition introduces a new schema called *RAddBirthday*, obtained by combining the three schemas on the right-hand side. The operation *RAddBirthday* must terminate whatever its input. If the input *name*? is already known, the state of the system does not change, and the result *already_known* is returned; otherwise, the new birthday is added to the database as described by *AddBirthday*, and the result *ok* is returned.

We have specified the various requirements for this operation separately, and then combined them into a single specification of the whole behaviour of the operation. This does not mean that each requirement must be implemented separately, and the implementations combined somehow. In fact, an implementation might search for a place to store the new birthday, and at the same time check that the name is not already known; the code for normal operation and error handling might be thoroughly mingled. This is an example of the abstraction

which is possible when we use a specification language free from the constraints necessary in a programming language. The operators $\wedge$ and $\vee$ cannot (in general) be implemented efficiently as ways of combining programs, but this should not stop us from using them to combine specifications if that is a convenient thing to do.

The operation *RAddBirthday* could be specified directly by writing a single schema which combines the predicate parts of the three schemas *AddBirthday*, *Success* and *AlreadyKnown*. The effect of the schema $\vee$ operator is to make a schema in which the predicate part is the result of joining the predicate parts of its two arguments with the logical connective $\vee$. Similarly, the effect of the schema $\wedge$ operator is to take the conjunction of the two predicate parts. Any common variables of the two schemas are merged: in this example, the input $name?$, the output $result!$, and the four observations of the state before and after the operation are shared by the two arguments of $\vee$.

RAddBirthday

$$
\begin{array}{l}
\Delta BirthdayBook \\
name? : NAME \\
date? : DATE \\
result! : REPORT \\
\hline
(name? \notin known \wedge \\
\quad birthday' = birthday \cup \{name? \mapsto date?\} \wedge \\
\quad result! = ok) \vee \\
(name? \in known \wedge \\
\quad birthday' = birthday \wedge \\
\quad result! = already\_known)
\end{array}
$$

In order to write *RAddBirthday* as a single schema, it has been necessary to write out explicitly that the state doesn't change when an error is detected, a fact that was implicitly part of the declaration $\Xi BirthdayBook$ before.

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

NotKnown

$$
\begin{array}{l}
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \notin known \\
result! = not\_known
\end{array}
$$

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

$$RFindBirthday \mathrel{\widehat{=}} (FindBirthday \wedge Success) \vee NotKnown.$$

The *Remind* operation can be called at any time: it never results in an error, so the robust version need only add the reporting of success:

$$RRemind \mathrel{\widehat=} Remind \land Success.$$

The separation of normal operation from error-handling which we have seen here is the simplest but also the most common kind of modularization possible with the schema calculus. More complex modularizations include *promotion* or *framing*, where operations on a single entity – for example, a file – are made into operations on a named entity in a larger system – for example, a named file in a directory. The operations of reading and writing a file might be described by schemas. Separately, another schema might describe the way a file can be accessed in a directory under its name. Putting these two parts together would then result in a specification of operations for reading and writing named files.

Other modularizations are possible: for example, the specification of a system with access restrictions might separate the description of who may call an operation from the description of what the operation actually does. There are also facilities for generic definitions in Z which allow, for example, the notion of resource management to be specified in general, then applied to various aspects of a complex system.