Practical exercises illustrating definitive representation of state

---

Download `roomviewerYung1991`. Launch the `tkeden` interpreter and execute the file `Run.e`. All this actually does is to load the contents of the file `roomviewer.s` into the interpreter. It will be useful to load the file `roomviewer.s` into a text editor so that you can inspect its contents. (On Linux, you can use a simple text editor such as `kwrite`. On Windows, `TextPad` is highly recommended for this purpose).

Looking at the screen display, you should be able to identify many aspects of its state that you can interpret. For instance, the window at the top-left contains something resembling the floor plan of a room. In a well-conceived script, we expect to find a close correspondence between features we can interpret, and observables on the LHS of definitions.

There are three components to the `roomviewer.s` script:

- Geometric elements relating to the features of the room floor plan
- Information about how the windows on the screen are disposed, what kind of information is displayed within them, and what properties they have.
- Identifiable state-changing actions that operate by performing specific redefinitions: these can be viewed as very simple *agents*.

These three components correspond respectively to three sections of the `roomviewer.s` script: lines 1-174, lines 175-367 and lines 369-467. Each section makes use of a different notation: essentially they are expressed in `donald`, `scout` and `eden` respectively.

Identifying and classifying the observables in the script

There are many ways in which you can explore the observables in a model. For instance: you can inspect the screen display and speculate on the possible interpretations of the features you see; you can think about any relevant domains to which the display might refer (in this example potentially properties of interest to an architect or room user); you can inspect the contents of the script; you can interrogate observables in the model. Whichever notation is used to define an observable, it's actual representation is translated into and stored in `eden` itself. The most comprehensive way to interrogate observables in the script is to type a query of the form:

```
%eden
?varname;
```

into the `eden` input window. To use this feature effectively, you need to know that a `donald` observable such as `desk/drawer/k` is represented within `eden` by the observable: `_desk_drawer_k`. You will also find it useful to understand something about how the values of points and lines in `donald` are represented in `eden`. Some sample queries are:

```
%eden
?_table_SW;

_table_SW is scalar_div((vector_add(_SW, _NE)), 2); /* current value of _table_SW is ['C
_table_SW ~> [P_table_SW, _table_S, _table_W, _table_SE, _table_NE, _table_NW, _table, _

?_SW;

_SW is cart(100, 100); /* current value of _SW is ['C',100,100] */
_SW ~> [P_SW, _S, _W, _SE, _NE, _NW, _table_SW, _desk_SW, _]; /* _SW last changed by inp
```

Note the list of observable names that appears after the ~> in the response to a query: this is the list of observables whose values are defined directly in terms of the queried observable.

A scout observable doesn't change its name on translation, but for complex types of observable (such as windows or displays) it is usually easier to make sense of the definition assigned in scout than its eden translation. Unlike donald, scout supports querying of observable names. By way of illustration, contrast the effect of querying the status of the window observable don1 (associated with the window at the top left of the screen display) using eden and scout:

```
%eden
?don1;

don1 is [1, [[0,0,100,100]], "", formbox(p1, q1), "view", DFxmin,
DFymin, DFxmax, DFymax, DFbgcolor, DFfgcolor, 1.0, DFalign, 1.0, DFbdcolor, DFfont, DFre
*/
don1 ~> [basicScreen]; /* don1 last changed by input */
```

```
%scout
?don1;

window don1 = {
    type: DONALD
    box: [p1, q1]
    pict: "view"
    border: 1
    sensitive: ON
};
```

Another way of finding out about observables in a model is to use the *View* option associated with the input window. For instance, it is possible to list the definitions of donald observables in the model.

Exploring the screen specification and layout

Notice that when you are thinking of the line drawing as a floorplan, you are subconsciously interpreting the screen display in a particular way. For instance, the boxes with text at the bottom are interpreted as buttons rather then (e.g.) as items that might be sitting outside the room ready to be brought inside. In this context, the conception of the state of the screen follows a standard pattern: the screen serves as a conventional interface, hence the room*viewer*. This "viewer" interpretation is reflected in the names given to the observables that make up the display.

In a scout script, the special observable screen records the state of the displayed screen. The observable screen is of type *display*, and consists of a list of windows that overlap in a specified manner. Each window is suitable for presenting a particular kind of content: the two windows at the top of the screen in the room viewer hold donald line drawings; those below contain text strings.

Using the query facility in scout it is possible to trace through the contents of the screen display. You begin by interrogating the display screen, to get:

```
display screen = if _cableIsShort then append(scr, 1, monCable) else scr endif;
```

You can then proceed to interrogate the observables scr, monCable, monDoor, basicScreen etc. By inspecting the definitions of these screen components, it is possible to find about all their attributes (location, content, background and foreground colours, sensitivity properties). Where these attributes are not explicitly specified, it is helpful to be aware of the defaults.

Particularly important in the case of scout windows of type DONALD are the pict field and the fields xmin, xmax, ymin and ymax:

- the pict field contains a viewport name that is associated with a set of definitions in a donald script.
- the fields xmin, xmax, ymin and ymax determine the precise region of the line drawing in the appropriate viewport that gets to be displayed. (This is specified using coordinates in donald.)

Exploratory interaction with the script

The above activities are all associated with "contemplation of a current state of the script" - a very important feature of Empirical Modelling. Interrogating observables doesn't essentially change the state of the script, but it does give access to otherwise hidden information. Other - more subtle and powerful - ways of exploring state are supported by atomic interactions, where the modeller tweaks the values of observables and sees what impact this has. These two kinds of ways of investigating state are broadly associated with 'observation' and 'experiment'.

You can change the current state by entering redefinitions through the tkeden input window. In this way, you can explore how each of the three components of the roomviewerYung1991 state (geometric features, screen layout, latent agency) described above is being modelled in the script.

Redefining geometric features

Interactions with the line drawing of the room were the subject of Labsheet 1. In that context, we considered some sample redefinitions of geometric observables:

```
%donald
door/open = false
plug = plug2
table/SW = {500,500}
desk/drawer/k = 1
```

These definitions are being considered here as independent, though there is nothing to prevent them being (conceptually!) executed "at the same time" if you wish.

In addition to the information about donald entity that is obtained by interrogating the variables name, there are also attributes of the entity that are recorded in an associated eden observable. For instance, the eden observable A_S records the attributes of the south wall of the room, so that (e.g.) its colour and width can be changed by making the redefinition:

```
A_S is "color=yellow,linewidth=3";
```

Note that there should be no spaces in the string that specifies attributes.

In addition to these modest tweaking of state, we can also make more ambitious redefinitions. For instance:

> *Revise the openshape table so that there is a new observable at the centre of the table, and all four corners are defined by an offset from the centre. Revise your definition so that the corners of the table are defined as offsets from the centre that are rotated by an angle tablerotangle about the centre. [You could also use polar coordinates for this.]*

Here's a possible solution:

```
%donald

## first create the table centre and define corners relative to it
within table {
        point centre
        real tablerotangle
        SW = centre - {width div 2, length div 2}
        NW = centre - {width div 2, -length div 2}
        SE = centre + {width div 2, -length div 2}
        NE = centre + {width div 2, length div 2}
}

## now rotate: initially I specified rotation "about the table centre"!
within table {
        point centre
        real tablerotangle
        SW = centre - rot({width div 2, length div 2}, {0,0}, tablerotangle)
        NW = centre - rot({width div 2, -length div 2}, {0,0}, tablerotangle)
        SE = centre + rot({width div 2, -length div 2}, {0,0}, tablerotangle)
        NE = centre + rot({width div 2, length div 2}, {0,0}, tablerotangle)
}

table/tablerotangle = 0.2
table/tablerotangle = pi div 4
```

Note: There are two mildly tricky issues to be resolved in order for the above revision to work. One is that the position of the `table/centre` needs to be defined by an explicit value at some stage. Ideally this should be done before any attempt is made to redefine `table/SW`, and - unless it is done "manually" - can only be done as an assignment in `eden` (there is no way to express an assignment in `donald`). The second issue is that the table moving buttons no longer work as they still act on `table/SW`!

Exploratory interactions with the screen layout

Similar kinds of redefinition, involving state-changes on various scales can be exploited in understanding the screen layout.

Simple changes might involve changing the location of a window, its background colour, or its content.

```
%scout
point p1, q1;
window don1 = {
    type: DONALD
    box: [p1, q1]               ## top left and bottom right corners - can relocate
    pict: "view"                ## line drawing content - can select another viewport
    border: 1                   ## thickness of border
    sensitive: ON               ## responsiveness to mouseclicks
};

%donald
viewport view
## where the definitions for the room model are listed
...
```

Any absent fields (see the Scout quick reference associated with the `Help` on the input window) have default values.

```
%scout
point p1, q1;

window don1 = {
    type: DONALD
    box: [p1, q1]               ## top left and bottom right corners - can relocate
    pict: "view"                ## line drawing content - can select another viewport
    border: 1                   ## thickness of border
    sensitive: ON               ## responsiveness to mouseclicks
    bgcolor: "white"
};
```

Alternatively, can define the background colour by a string, as in:

```
%scout
point p1, q1;
string bgcol;
window don1 = {
    type: DONALD
    box: [p1, q1]                 ## top left and bottom right corners - can relocate
    pict: "view"                  ## line drawing content - can select another viewport
    border: 1                     ## thickness of border
    sensitive: ON                 ## responsiveness to mouseclicks
    bgcolor: bgcol
};
bgcol = "white";
```

Observables such as `bgcol` can then be assigned definitions or explicit values using either `scout` or `eden`.

Note that the order in which `scout` windows are listed in the `screen` display determines which window appears on top of another. This order can be manipulated both in `scout` and `eden`.

State changing actions associated with simple agents

The most primitive way of interacting with a model is to change the state directly yourself by entering redefinitions via the input window. We can model another way in which state-changes might be effected, by delegating some other agent to make redefinitions on our behalf. This is illustrated by the buttons in `roomviewerYung1991`. We can think of the potential for state-change as a form of agency implicit in a situation - something may be capable of acting, or (in some contexts) currently be acting.

Simple agency in the viewer interface is mediated by observables that are associated with `scout` windows that are specified as sensitive. (Most of the windows in the room viewer screen are sensitive - you can check their definitions to see how this is specified.) There is an observable `don1_mouse` associated with the `don1` window for instance, and an observable `doorButton_mouse_1` associated with the `doorButton` window. (The "_1" suffix here is required because in principle a window containing text can be made up as a list of several boxes, so that the text content runs from one box to another. This feature is very rarely used, and may not be correctly supported in all versions of `tkeden`.)

To see that clicking the mouse in these windows affects the state of these observables, you need to select "View history" in the *View* menu associated with the input window. Clicking in the `don1` window causes a list of values to be assigned to the observable `don1_mouse`:

```
don1_mouse = [1,4,0,378.109452736, 517.412935323];
```

The values on the right hand side here indicate which mouse button has been pressed [1, ...], the fact that it is depressed [.., 4, ...], a qualifying mode determined (e.g.) by whether shift or control is also depressed [..., 0, ...], and coordinates of the `donald` point at which the line drawing has been clicked [..., 378.109452736, 517.412935323]. To find out how the mouse click is linked to an action, you can find out what the mouse click triggers by querying the observable:

```
don1_mouse=[1,5,256,378.109452736,517.412935323];
don1_mouse ~> [don1_to_tableSW, don1_mouseButtonPress];
/* don1_mouse last changed by interface */
```

Redefining the observable `don1_mouse` triggers the execution of the `eden` action:

```
proc don1_to_tableSW: don1_mouse {
    auto mx, my;
    ...
}
```

The core of this action is a redefinition of the `table/SW donald` observable, which is encoded in `eden`:

```
_table_SW = ...
```

Interpreting this action is a useful exercise in understanding how state changes most easily expressed in `donald` can be carried out in `eden`.

Further notes on technical points

1. The observables in `donald` are interpreted in `eden` using definitions, functions and actions. For instance,

a `donald` definition of the form:

```
within table {
        point SE, SW
        int width
        SE = SW + {width, 0}
        ...
}
```

generates corresponding observables in `_table_SE`, `_table_SW` and `_table_width` in `eden`, and links these by a translated *definition*:

```
_table_SE is vector_add(_table_SW, cart(_table_width, 0));
/* current value of _table_SE is ['C',800,500] */
```

The scalar components of points and lines are recorded by lists of scalars in `eden`. The `eden` *function* `vector_add` is the counterpart of vector addition of points in `donald`. By querying the status of `_table_SE`, you can also detect the presence of an *action* `P_table_SE` which is triggered every time the point `_table_SE` is redefined:

```
_table_SE ~> [P_table_SE, _table_S, _table_E, _table];
```

This illustrates a standard technique for translating definitive notations into `eden` that accounts for its identity as an **e**valuator for **de**finition **n**otations

2. You might be unhappy about the way in which the scale of the floorplan is expressed in terms of absolute coordinates. This can be fixed if you wish by modifying the way in which the `donald` picture is disposed in the `scout` window (see note 4 below). If you want to establish the coordinates of a `donald` picture in a `scout` window (and the window is specified as sensitive to mouse clicks), you can select *view history* on the `View` menu of the `tkeden` input window, and observe the values assigned to the observables to the mouse when it is clicked in the window. You can confirm that the coordinates of the floor plan range from {0,0} at bottom-left to {1000,1000} at top right by clicking in the window and noting that:

```
      don1_mouse = [1,5,256,960.199004975, 965.174129353];
```

near the top right hand corner etc.

3. For polar coordinates in `donald`, you can use definitions such as:

```
point ppolar
real modulus, theta
ppolar = {modulus@theta}
modulus = 600.0
theta = pi div 4
```

and test this out by introducing a definition such as:

```
table/SW = ppolar
```

4. The way in which a `donald` picture is disposed in a `scout` window makes a small case-study in itself. Relevant features to understand are the role of "viewports" in `donald`: you can explore this by examining the `donald` definitions and looking at the `pict` tags in the `don1` and the `don2` windows. There is then a subtle interplay between the location of the `scout` window (examine the points `p1` and `q1` which specify locations in terms of pixels from the top left hand corner of the screen) and the values assigned to the fields `xmin`, `xmax`, `ymin` and `ymax` within the definition of the `scout` window. In order to preserve the aspect ratio of a `donald` picture you must specify the same ratio of length to width for the `scout` window and for the region of the `donald` picture being displayed.