A functional reactive animation of a lift using Fran

Simon Thompson
Computing Laboratory, University of Kent
Canterbury, Kent, CT2 7NF, UK
(*e-mail:* `S.J.Thompson@ukc.ac.uk`)

Abstract

This paper uses the Fran system for functional reactive animation to give a simulation of a lift – or elevator – with many floors. The paper first introduces a two-floor version, and then indicates in detail how this is extended to give a simulation with an arbitrary number of floors and featuring more realistic animated graphics.

The paper introduces those aspects of Fran relevant to the simulation, making it a self-contained tutorial on parts of Fran and how it is applied in practice. The full code for the system is available on the World Wide Web.

1 Introduction

This paper uses the Fran system for functional reactive animation (Elliott & Hudak, 1997; Peterson *et al.*, 1997) to give a simulation of a simple lift or elevator. Fran is a substantial library extending the Haskell (Hughes & Peyton Jones, 1999) functional programming language on Windows platforms. The work discussed here has been developed using the Hugs interpreter (1998); compiled support is available using the Glasgow Haskell Compiler (1998). The main architect of the Fran system is Conal Elliott of Microsoft Research, whose earlier work used C++ as a vehicle for similar ideas (Elliott *et al.* , 1994).

The functional approach of Fran is justified by the fact that the authoring medium for animations ought to "… give the author complete freedom of expression to say what an animation is, while invisibly handling details of discrete, sequential presentation. In other words [it] must be declarative …"(Peterson *et al.*, 1997). This approach is familiar to the functional programmer; one can see it in influential work on 'functional graphics' (Henderson, 1982) some fifteen years ago as well as in more recent approaches to describing music in a declarative form (Hudak *et al.* , 1996), to name but two examples.

Fran provides two complementary modelling abstractions. First, `Behavior X` is the type of *time-dependent* values of type `X`. A time-dependent image is a graphical animation, for instance. On their own these behaviours are effectively predetermined: once initiated they evolve autonomously. In order to react to internal or external events of various sorts, Fran provides `Event` types, which can model, for instance, user input, timers, and signals between concurrent components of an animation.

In this paper the Fran system is introduced in stages, and this is interleaved with a description of a version of the lift problem in Section 3.1 together with a top-down description of the lift simulation itself in Sections 3.3, 4.2, 5.1, 6.2 and 6.3. After completing the two floor case study, we examine in Section 8 how the system is extended to accommodate an arbitrary number of floors, and also give an overview of the animation of various graphical aspects of the system.

This is followed in Section 9 by a brief 'look under the bonnet' to see some of the primitives used to define the operators used in the case study; this is followed by some reflections on Fran and how Fran programs might be verified in a temporal logic, the topic of a companion paper, (Thompson, 1999).

The introduction to Fran given here is intended to make the paper self-contained; a more comprehensive introduction is available in the papers cited above and in animated form at (Elliott, 1997).

It is interesting to observe the positive benefits of embedding Fran in the declarative framework of the Haskell programming language. Beyond providing a natural home for a declarative modelling tool, the library is able to exploit features such as polymorphism and type classes. In writing this simulation we also have been able to exploit the power of the language in building general 'terminating' simulations (Section 10) and in writing general building-blocks for graphical interface components (Section 7).

It is also interesting to observe the beneficial effect of working in a typed environment: particularly when working with the libraries for `Events`, it was often possible to find the right component of the library by its type. Moreover, in nearly all cases, if a piece of code passed the type checker it was correct. It is all too easy to imagine what would happen in an untyped or weakly-typed language.

I am very grateful indeed to Conal Elliott who has answered my numerous questions, made suggestions about improvements to programming style and generally encouraged me in this enterprise. Erik Poll has made a number of very useful comments on the presentation of the material, and the referees have made a number of helpful suggestions. Finally I would like to thank Howard Bowman, Helen Cameron and Peter King for their collaboration in our work on describing multimedia artifacts which has led to my looking at Fran.

2 Behaviours

This section looks at the way that continuously-evolving behaviours can be described in Fran.

2.1 Time-dependent values: `Behaviors`

Behaviours or time-dependent values of type `X` are given by the type `Behavior X`. These can be thought of as functions of type

```
Time -> X
```

where `Time` is the domain of real numbers.

In fact, the representation of `Behavior` will be more complex for two reasons. First, the domain `Time` is more complicated than simply the reals for reasons discussed in (Elliott,

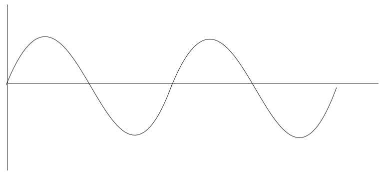


Fig. 1. The behaviour `wiggle`

1998). Secondly, it can be more effective to work with *representations* of functions rather than with functions themselves. In both cases, however, the simplified model of behaviour suffices for our purposes here.

How are `Behaviors` implemented? Full details can be found in (Elliott & Hudak, 1997; Elliott, 1998) but a simple-minded model is to think of animations as being given by sampling the values of functions at a series of points; under this model an animated image would be made up of a sequence of distinct images generated at the sampling points. This is not, in fact, how all Fran behaviours are implemented; images, for instance, are implemented in a more efficient manner.

Among the types we shall use in our solution are

```
type RealB  = Behavior Double
```

and `ImageB`. An example value of type `RealB` is the `wiggle` illustrated in Figure 1. which is defined by

```
wiggle = sin (pi * time)
```

using the built in `time` of type `RealB`, which can be thought of as the identity function.[1]

Primitive graphical objects include circles, rectangles, polylines, polygons and so forth. Pure graphical animations can be built from these and various library functions, some of which are discussed now.

`moveXY :: RealB -> RealB -> ImageB -> ImageB` [2] The first two arguments give the x- and y-coordinates of the position that the `ImageB` should take at each time. Note that it is not simply an image that is moved; it is an `ImageB` which can itself be moving, changing shape or colour and so on.

`over :: ImageB -> ImageB -> ImageB` This supports the superimposition of the first image over the second, giving a form of concurrency — we discuss this further in Section 5.2.

`stretch :: RealB -> ImageB -> ImageB` This scales an `ImageB` according to a `RealB`, so allowing the size of animated objects to vary with time.

`withColor :: ColorB -> ImageB -> ImageB` The effect of this function is to change the colour of an `ImageB` according to the time-dependent colour supplied.

[1] Note here how the type class mechanism of Haskell supports overloading and in this case the use of `sin` and `pi` over `RealB` rather than `Double`. This overloading makes Fran programs substantially more readable.

[2] The double colon, `::`, should be read 'is of type', and the type here is a function taking three arguments. Functions in Haskell are in fact 'curried' so that strictly `moveXY` takes a single argument and returns a function as result.

An example behaviour is given by

```
moveXY wiggle 0 pic1
`over`
moveXY 0 waggle pic2
```

in which `pic1` "wiggles" from left to right, `pic2` "waggles" (cosine) up and down and `` `over` `` is the infix form of the function `over`, and so superimposes the two `ImageBs`, which are themselves moving and may indeed be evolving in other ways.

Using these functions it is possible to build graphical animations of the non-reactive aspects of a lift simulation, such as graphics of a lift whose doors are opening, a lift whose doors are closing, a lift in motion and so forth, since these are built from blocks of colour of changing size and position.

2.2 Rate-based animation

Suppose we want a numerical quantity `f` to change with time as part of an animation. One way of doing this is to specify `f` as a function of time, as indeed we did with `wiggle` above. Fran provides an alternative, by which we specify the rate of change (or derivative) of the quantity, `f'` say, and using the `atRate` function `f` can be given as the integral of `f'`.

A simple example is given by position and velocity: a linear change in position is given by a constant velocity, for instance. This example reflects the general observation that it is often easier to describe the derivative of a function rather than the function itself, and we shall see an example of this in our simulation.

3 The case study: lift simulation

This paper addresses the first of the problems set for a workshop on 'Challenges for Executable Temporal Logics' (ETL, 1998), namely that of simulating the operation of a lift (or elevator). The problem as originally stated requires an arbitrary number of floors; in this paper we approach the problem by giving the full solution for the two floor case, and then by giving a detailed overview of the general case. In the introduction we have also stripped down the graphics to concentrate on the control aspects of the problem, which in the case of Fran are the aspects demanding the most effort to implement.

In our first attempt we control the operation of the lift using the mouse buttons. We show how to modify our solution to include a more general graphical control scheme in Section 7 and Section 8 explores the n-floor problem.

3.1 The two floor problem

The scenario for the two floor lift simulation is as follows.

> The aim is to provide a graphical animation of the operation of a lift between two floors of a building. The lift can be called from the upper floor to request travel downwards; a call from the lower floor is taken to be a request to travel upwards. Because of this interpretation, there is no need explicitly to model buttons inside the lift itself. Input is taken from the mouse buttons: a left button click generates an 'up' request and the right a 'down'. In the solution presented, the lift is represented by a red blob, rather than any more complicated animation.

There is a twist to the problem which makes its solution more complex than might be first envisaged. This is the fact that while the lift is travelling upwards there can be a further request to travel upwards, which can only be discharged by a journey back down and then back up again. In other words, some "memory" is required to solve even the case of a two-floor simulation, and this shows that our simplification contains most of the essential elements of the n-floor problem.

3.2 The `User` argument

Any animation which uses time information in a non-trivial way or which interacts with the user will be defined as a function which takes a `User` argument and, on the basis of this argument, returns a `Behaviour` of some sort. The `User` argument consists of a timed stream of mouse button presses, key presses, mouse positions and other user data. It also gives the time at which the animation begins and other real-time information.

In order to exhibit such a user-driven animation in action we use the Fran function

```
displayU :: (User -> ImageB) -> IO ()
```

which 'runs' an animation, by supplying it with the user input stream as its `User` argument, and produces primitive Haskell IO as a result.

3.3 Case study part 1: the top-level solution

We shall give the solution to the lift problem top-down, with the full code for the solution appearing in the Appendix. At the top level we define a function over a `User` argument, as explained in Section 3.2:

```
liftSim :: User -> ImageB
```

The simulation consists of a moving image of a lift,

```
liftSim u
  = moveXY xPos yPos image
```

where `xPos` and `image` are constants.[3] The definition of `yPos` and the auxiliary behaviours and events which determine it are local to the definition of the function and thus appear in a `where` clause. Note that the result depends upon the `User` argument `u` which will be used in a number of the definitions which follow.

As we explained in Section 2.2 it is often simpler to model phenomena by derivative rather than directly, and this we do here for the `yPos`:

```
yPos = lower + atRate dy u
```

where the velocity – `dy` – is piecewise constant and can take one of three values: zero, making the lift stationary; the positive value `upRate`, signifying that the lift is ascending and the negative value `downRate` for descent. We have to look at how events are modelled to see how `dy` is defined and this we do in the next section.

[3] In a more complex simulation `image` would itself evolve. Its definition would follow the pattern of that for `yPos`; more details are given in Section 8.

Observe also that the `User` argument `u` is passed to the `atRate` function to provide the timing information – such as when the animation begins – needed by the integration.

4 Events

We have seen in Section 2 how certain simple time-dependent behaviours can be defined, but in order to define behaviours which respond to internal conditions or external events the model needs to be extended by the `Event` type.

An `Event X` is a sequence of timed occurrences, each of which is associated with a value of type `X`, so it is possible to think of `Event X` as a list of type

```
[(Time,X)]
```

sorted on its first components. Each of the elements, `(t,x)` say, is called an **event occurrence**, with the whole structure being the event.[4] (As was the case for behaviours, the implementation is somewhat more complicated than this, but for the purposes of this paper this will suffice.)

4.1 Handling events

The system provides substantial support for handling and modifying occurrences of events. In our model we only perform simple transformations on events.

Each event occurrence in a stream `str` of type `Event a` will have the form `(t,x)`, where `t::Time` and `x::a`. In every case we are interested in as a part of the case study, our aim will be to convert the value `x` to a fixed value `c::w`, thus converting the pair to `(t,c)`. The resulting stream will be written

```
str -=> c  :: Event w
```

The effect on a stream `[(t0,x0),(t1,x1),...,(tn,xn),...]` is therefore to produce the result `[(t0,c),(t1,c),...,(tn,c),...]`.

In the general case, the event occurrence produced by the event handler corresponding to `(t,x)` may depend upon `x`,[5] `t` and the remaining part of the `Event` (after the expired event occurrences are removed). Further details of the event handling mechanism `handleE` can be found in Section 9 and (Peterson *et al.*, 1997).

Streams of event occurrences can be merged, time-wise, using the operation

```
.|. :: Event a -> Event a -> Event a
```

These two capabilities allow us to proceed further with our lift case study.

4.2 Case study part 2: top-level events

The movement of the lift is controlled by a number of `Events` generated internally

[4] This terminology appears to be in conflict with the more usual 'event' (for 'event occurrence') and '(event) stream' (for 'event'); we will use the Fran terminology in this account.
[5] In which case the operation is similar to the `map` function over lists.

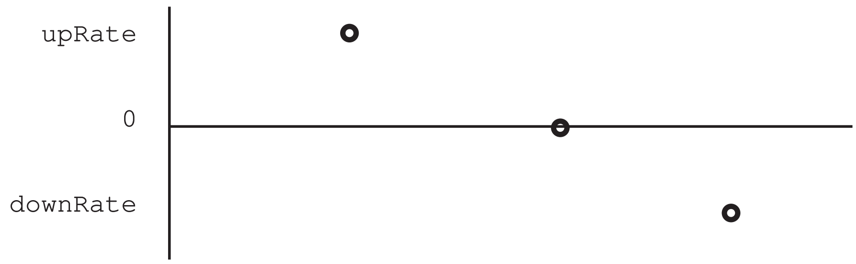


Fig. 2. An example event

```
stop, goUp, goDown :: Event ()
```

which are intended to give the obvious values to the velocity of the lift, `dy`. `()` is the trivial type, whose only member is also denoted `()`; it is used in a situation where the value contained in the `Event` occurrence is of no significance and only the `Time` value is relevant, as is the case here. The `Events` will themselves be defined in Section 6.3.

The definition of `dy` uses `setRate :: Event Double` given by

```
setRate = stop   -=> 0
          .|.
          goUp   -=> upRate
          .|.
          goDown -=> downRate
```

The effect here is to convert all occurrences of `stop` to occurrences with the value `0`; all occurrences of `goUp` to event occurrences with the value `upRate` and so on. These are merged, and so give an `Event Double` of the form

```
[ (2.3 , upRate) , (4.9 , 0) , (8.6 , downRate) ...  ]
```

in which `upRate` is the value returned at the occurrence at time `2.3` and so forth. This is depicted in Figure 2. In order to define `dy` this timed stream of values needs to be converted to a piecewise-constant behaviour and this we investigate now.

4.3 Converting `Events` to `Behaviors`

Behaviours evolve in continuous time, while (occurrences of) events take place at discrete points in time. The power of the Fran model lies in the way in which these two types are linked. This section addresses one simple way in which `Events` can be converted to `Behaviors`; other more complex (and more fundamental) ways are examined elsewhere (Peterson *et al.*, 1997). We need to convert the event `setRate` to a behaviour. This is done by the Fran function

```
stepper :: a -> Event a -> Behavior a
```

which is parametrised by a starting value and an `Event`, and builds a piecewise constant `Behavior` from these inputs. With the starting value `0` and the event as above we have the behaviour illustrated in Figure 3, so we define

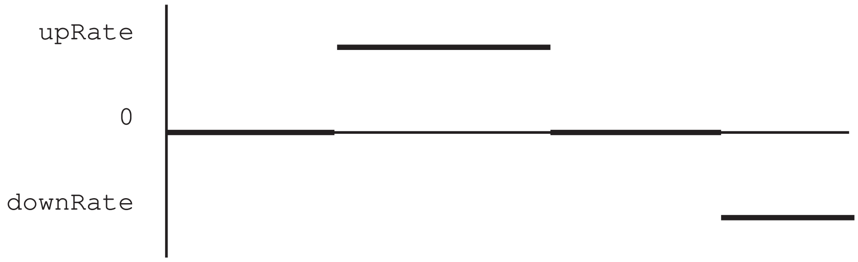


Fig. 3. An example behaviour defined from an event.

```
dy = stepper 0 setRate
```

This shows the mechanism by which we convert streams of ‘internal messages’ – `goUp` and so forth – into a behaviour, and completes the top-level definition of the simulation. We now need to look at how these messages are themselves defined, but before that we investigate how to model system states.

5 Modelling states in Fran

As we mentioned in Section 3.1, the implementation of the lift will need to contain some element of memory to keep track of requests for travel that are still to be fulfilled. In an earlier version of the solution, a state monad (Wadler, 1995) was used to model the state, but this caused complications, particularly in conjunction with handling a `User` argument (which could itself be seen as giving rise to a monad).

There is a much more straightforward view of a state variable of type `X`, and that is as a `Behavior X` – an `X` value which varies with time. We thus get a declarative model of state in Fran.

5.1 Case study part 3: pending requests as ‘variables’

Our model contains three ‘Boolean variables’

```
upPending, downPending, pending :: Behavior Bool
```

which keep track of whether there is pending a request to travel up, down or in either direction. `pending` is the pointwise disjunction of `upPending` and `downPending`

```
pending = upPending ||* downPending
```

Here `||*` is the lifting of the Boolean disjunction operation `||` to `Behavior Bool`; other operations such as == are lifted to behaviours in a similar way below.

The state `upPending` is defined from an `Event` using `stepper` as above.

```
upPending = stepper False (setUp .|. resetUp)
```

The initial value of the variable is `False`; it is set to `True` by a request to travel upwards, and reset to `False` by the lift starting an upward journey, as signalled by `goUp`:

```
setUp   = upButton -=> True
resetUp = goUp     -=> False
```

The state variable `downPending` is defined in a similar way.

5.2 Concurrency in the Fran model

The model presented thus far appears to contain elements which evolve concurrently, in some sense at least. This section attempts to explain the nature of the concurrency in the system.

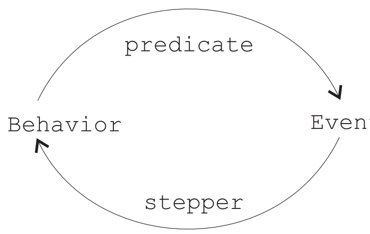
Animations – including sound as well as images – can be combined using the `` `over` `` function which places one animation on top of another, so we have a form of concurrency here.

Examining the implementation developed thus far, we appear to have `Behaviors` evolving in parallel in the lift simulation: the system contains state variables which are controlled by messages from other parts of the system, for instance. The concurrency here is completely implicit: various interdependent values – both `Behaviors` and `Events` – are defined in a single scope, and so can be thought of as evolving concurrently. This concurrency is clearly supported by the sample/display model which underlies Fran.

Under this interpretation of Fran, event occurrences can be seen as signals which communicate between concurrently-evolving behaviours.

6 Predicates

Looking at the case study thus far, we have a model of the lift in which external stimuli are provided by mouse button presses. We need, however, to find a way of generating the 'control' messages, `goUp` and so forth – which are of type `Event ()` – from the `Behaviors` in the system. The way in which we convert from behaviours to events gives the last piece of the Fran model used here.



6.1 The `predicate` function

In order to complete the simulation, the crucial pieces of information which we need are when we have arrived at the top or the bottom of lift shaft. We could keep an internal record of the time of departure and calculate offsets from that, but here we choose instead to check when we have arrived by means of a logical condition on behaviours. This `Behavior Bool` can be made to generate an event by means of the function

```
predicate :: BoolB -> User -> Event ()
```

which takes a Boolean behaviour and the `User` argument (for timing information) and returns an `Event`. We have to look for an appropriate `BoolB` with which to test having arrived at the top. Candidates include

`yPos ==* constantB upper` Given the sampling model, it is possible that the system will miss the point at which the condition is `True`. A fuller discussion of this issue can be found in (Elliott & Hudak, 1997), which makes it plain that implementations of `predicate` have changed with different releases of Fran.

`yPos >=* constantB upper` This condition will possibly be `True` over an interval of time, or in a more problematic way may become `True` arbitrarily often over a short period of time, giving rise to "event burst" as it were. This was indeed a problem in an early version of the solution.

`yPos >=* constantB upper &&* dy >* 0` Adding the condition that the lift is in motion – `dy >* 0` – gives this condition a *transitory* property: we shall see in Section 6.3 that the `Event` to which it gives rise will ensure that it becomes `False` immediately afterwards, thus avoiding the problems of the previous possibility.

We can now put together the final parts of the solution of the case study.

6.2 Case study part 4: conditions and predicates

The conditions of being at the top, and waiting stationary (at the top) are given by

```
atTop, stopped, waitingTop :: BoolB
atTop      = (yPos >=* upper)
stopped    = (dy ==* 0)
waitingTop = atTop &&* stopped
```

The `Event` of arriving at the top is defined by

```
arriveTop :: Event ()
arriveTop = predicate (atTop &&* dy >* 0) u
```

and similar definitions can be found for the bottom case.

6.3 Case study part 5: control `Events`

Recall that the top level of the simulation is driven by the 'control' `Events` `goUp`, `stop` and `goDown`. Now that we have a definition of the `Event` generated by arriving at the top (and bottom) we can define the control events.

We have seen earlier that button presses have an indirect effect on the operation of the lift by setting the pending variables; they can also have a direct effect by setting it into motion when it is stationary. We therefore define

```
upButton, downButton, eitherButton :: Event ()
upButton     = lbp u
downButton   = rbp u
eitherButton = upButton .|. downButton
```

so that the `eitherButton` event corresponds to the press of either button.

When should the lift go down; that is, when should there be occurrences of the `goDown` event? There are two cases.

- The lift can arrive at the top (`arriveTop`) when there is a request pending for travel in either direction (`pending`), or,
- either button can be pressed (`eitherButton`) while the lift is waiting at the top (`waitingTop`).

In both cases there is a condition on the event occurrences. Using the function

```
whenE :: Event a -> BoolB -> Event a
```

the expression

```
ev ‘whenE‘ cond
```

selects from `ev` precisely those occurrences at which the condition `cond` is `True`.[6] The definitions of `goDown` and `stop` are then

```
goDown = arriveTop    ‘whenE‘ pending
         .|.
         eitherButton ‘whenE‘ waitingTop

stop   = (arriveTop .|. arriveBottom)
         ‘whenE‘ notB pending
```

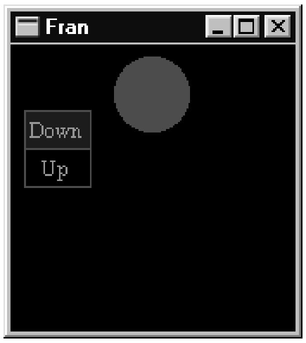
It is not hard to see that the lift will stop in the situation of arriving either at the top or the bottom when no request is pending. `goUp` is defined by analogy with `goDown`.

This completes the definition of the Fran lift simulation for a two floor lift. The full code is contained in the Appendix.

7 A graphical interface

This section outlines the way in which the implementation can be given a graphical interface.

The lift is controlled by the mouse buttons in the solution presented thus far. We can modify this to include two on-screen buttons, as in the illustration



[6] `whenE` acts rather like the standard function `filter` over lists.

which shows the lift in motion with a pending request for travel downwards. This modification is made with minimal changes to the code presented thus far. As well as having to modify the code already written, we have to add to the code a block of buttons and for this we use a function `buttonBlock` of type

```
Geom ->
[(String,Event (),a)] ->
(User -> Event ()) ->
User ->
(ImageB,Event a)
```

where the arguments consist of:

- the geometry of the buttons;
- information for each button:
  — its textual form,
  — the event which resets it, and
  — the value identifying it as having been pressed in occurrences of the `Event` generated by the button block;
- the event which presses the button;
- the `User` input.

The result consists of the image of the block paired with the `Event` which it generates.

This function is not part of the Fran library, and it is noteworthy that it can be implemented from scratch in about a hundred lines of code. This is a benefit of embedding the Fran library in a general purpose, higher-order language like Haskell.

Taking input from on-screen buttons is obviously a way of animating the input for the n-floor problem, to which we turn now.

8 The n-floor problem

This section gives a design overview for the n floor lift and *inter alia* suggests a taxonomy by which the elements of a Fran program can be classified. The fully-commented code can be downloaded from

`http://www.cs.ukc.ac.uk/people/staff/sjt/Fran/`

8.1 The scenario

The scenario for the n-floor problem is described thus.

> The lift has floors numbered from `0` to `topFloor`. Requests to travel to these floors are given by typing a single digit at the keyboard.[7]

[7] This is a non-essential simplification of the case in which requests to travel up or down are made at each floor, whilst specific floor choices are made in the lift. Modelling this requires an extra component in the state and is left as an exercise for the reader.
It is also a straightforward exercise to build a more flexible input component which processes input by button presses, say, and which would model a lift with more than ten floors.

The lift changes direction of motion only when this is necessary: if the previous direction of motion was upward, then, after stopping at a floor, the lift will continue upwards if there are any outstanding requests to visit floors higher up.

This is a simple expedient to ensure that all requests are eventually discharged, assuming only that the lift is not held at a particular floor by someone repeatedly re-opening the doors before they close. A strategy of always going to the closest floor irrespective of direction would not necessarily process all requests in all circumstances.

On arriving at a floor to which a visit is pending, the lift doors open and then close to discharge and receive passengers. Should a request to travel to that floor be received during this process, the doors re-open from their current position and close again.

The remainder of this section presents an overview of the n-floor lift, first looking at its vertical motion in Sections 8.2–8.7 and then at the graphical animation of its doors in Section 8.8.

8.2 Vertical motion: basics

Exactly as in the two floor design, the vertical motion of the lift is modelled using its vertical velocity, `dy`, which is integrated to give the vertical position `yPos`. Moreover, `dy` will be controlled using the `stepper` applied to the event `setRate`, which is itself defined as before from the events `stop`, `goUp` and `goDown`. It is the re-definition of these events solves the n-floor problem.

The program which models the lift is made up of behaviours and events; it is useful to further classify these components.

8.3 Continuous and piecewise constant behaviours

Behaviours in the lift model come in two sorts. **Continuous** behaviours model *physical quantities*, like the vertical position of the lift (`yPos`) and the position of the edge of the doors (`doorEdge`). Other behaviours are **piecewise constant**, as illustrated earlier in Figure 3, and these correspond to various *state variables* of the system. In this model the state variables include

`currentFloor`	`Int`	the 'current floor' or, when the lift is moving, the last floor to have been passed or visited by the lift;
`pending`	`[Int]`	the (ordered) list of floors to be visited;
`lastUp`	`Bool`	the value indicating whether or not the last vertical motion of the lift was upwards;
`dy`	`Double`	the vertical velocity of the lift.

8.4 Controlling state variables

In the two floor solution, the state variables such as `upPending` have only two values. This means that when they are reset the new value is independent of the current value of the state variable. This will not be the case for the state variables here; for instance, the value of `currentFloor` will be incremented or decremented by one on reaching the next floor above or below the previous one.

Instead of using the `stepper` to define a behaviour from an event, we will use the function

```
accumCB :: (a -> b -> a) -> a -> Event b -> Behavior a
```

to manage the states.

The crucial argument to `accumCB` is the function of type `a -> b -> a`; this function is used to generate the new value of the state from the current value of the state variable (of type `a`) and the value of type `b` contained in an event occurrence.[8]

For example, the `currentFloor` is defined by

```
currentFloor :: Behavior Int

currentFloor = accumCB (\n b -> if b then n+1 else n-1) 0 arrival
```

which defines the `currentFloor` which is incremented or decremented depending upon the Boolean value returned by the `arrival` event, signalling whether the arrival is in an upward or downward direction.

A second example is given by the list of `pending` floors. A floor is added when the corresponding key is pressed; the first argument to `accumCB` in this case will be a function to insert a floor number into the ordered list of floors to be visited.

8.5 External and internal events

The lift simulation contains two different kinds of event. It is a reactive simulation and so one class of events are **external** and are produced by the user of the simulation. In this particular case they will be key presses, which are translated into the event

```
selectFloor :: Event Int
```

in which the value contained in an event occurrence is the floor requested.

Events provide a way of determining behaviours by means of the `stepper` and `accumCB` functions. The external event of selecting a floor will affect the state variable `pending`, but other effects are produced by events which are internal to the system.

An event is **internal** to the system if is it generated from other events or behaviours in the system. Among the most important are the **primitive events** generated by certain conditions becoming true, using the `predicate` operation of Fran. In the n-floor lift these include

`arrivalUp`, `arrivalDown`	`Int`	arriving at a floor, travelling up or down; the value returned is the floor reached;
`doorOpen`, `doorShut`	`()`	the door becoming fully open or fully shut.

Other internal events are derived from other events; events can be merged, can have occurrences selected according to predicates,[9] and can have values transformed by `-=>`. The derived internal events include

[8] The function `accumCB` is analogous to the `scanl` function from the Haskell prelude.
[9] akin to the `filter` function of Haskell

`setRate`	`Double`	the event determining the vertical velocity; which uses the events which follow …
`goUp, goDown, stop`	`()`	… events governing changes in vertical motion
`arrival`	`Bool`	arriving at a floor; the value `True` indicates arrival from below.
`waitingDown, waitingUp`	`()`	events signalling the transition from waiting to moving down or up;

8.6 `Event` definitions

The main events in the model are arrival at a floor and `stop`, `goDown` and `goUp` which determine changes in vertical velocity.

To determine arrival at a floor, we use the `predicate` operation and define `arrivalUp` of type `Event Int`

```
arrivalUp
  = snapshot_ (predicate ( yPos >=* newFloorPosition &&* dy >* 0) u) newFloor
    where
    newFloor         = currentFloor + 1
    newFloorPosition = lift1 floorPosition newFloor
```

The `currentFloor` variable keeps a record of the last floor to be visited or passed; on the basis of this it is possible to calculate the `newFloor` and its position, `newFloorPosition`. Arrival at the floor is signalled by an inequality, exactly as in Section 6.1.

A novel feature here is the Fran function `snapshot_` which captures the floor number by taking a 'snapshot' of the value of `newFloor` at the time of arrival, making this the value returned by the event occurrence.

When is the lift supposed to stop? When it reaches a floor which is an element of the pending list: this is expressed clearly by the definition

```
stop   = whenSnap arriveFloor pending elem -=> ()
```

The lift will start to go down under two circumstances,

```
goDown = doorShut `whenE` downCond
         .|.
         waitingDown
```

represented by a merge of two events.

- The first occurs when the doors shut and the condition for downward motion holds. The condition for going downwards, `downCond`, is a Boolean behaviour calculated on the basis of the state variables which record the current floor, the pending floors and the last direction of motion.
- The second occurs when a button is pressed for a floor below the current one and the lift is waiting:

```
waitingDown
  = whenSnap selectFloor currentFloor (<) -=> () `whenE` waiting
```

The corresponding events for upward motion are defined in a similar way.

8.7 Inter-component dependencies

It is instructive to examine the dependencies between the various events and behaviours which make up the lift simulation. An analysis shows that almost all the behaviours and internal events are inter-dependent – technically they form a single strongly connected component of the function call graph – with only the external event of floor selection being independent of them.

This information may go some way to show why the system can be somewhat slow in execution. The simulation is a tightly-coupled system in which it is not possible to divide the functionality into rather more independent components.

8.8 The doors

At the top level of the program for the n-floor lift the lift image is given by

```
moveXY xPos yPos image
```

but in contrast to the two floor case, the `image` is itself animated:

```
image = frame `over` doors
```

The `frame` is a constant image, whilst the `doors` open and close. The position of the doors is given by integrating their velocity, which is itself a `Behaviour` given by `stepper` from the event `changeDoorVel`.

```
changeDoorVel
  = doorOpen -=> (-vel)
    .|.
    doorShut   -=> 0
    .|.
    openDoors -=> vel
```

The events `doorOpen` and `doorShut` signal when the doors become fully open or fully shut; they are defined using a `predicate` over the position of the doors, like the way in which arrivals were defined earlier by reference to the vertical position of the lift.

The event `openDoors` signals when the doors are to be opened.

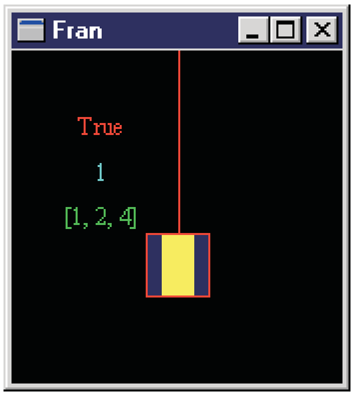
```
openDoors
  = whenSnap arriveFloor pending elem   -=> ()
    .|.
    (whenSnap selectFloor currentFloor (==)) `whenE` stationary -=> ()
```

This definition has the effect that they are opened either

- when the lift arrives at a pending floor, or
- when the current floor is selected (the `whenSnap`) and the lift is stationary (given by the `` `whenE` ``).

In the latter case, this mean that doors are re-opened if the ‘up’ button is pressed while the doors are closing.

On execution the simulation will appear as



in which the current values of `lastUp`, `currentFloor` and `pending` are also shown for diagnostic purposes. In the figure, the doors are closing on level 1, with levels 1, 2 and 4 to be visited, and the previous direction of motion having been upwards (indicated by the Boolean value `True`). As soon as the doors close, the value 1 will be removed from the pending list.

8.9 Conclusion

The development of the n-floor lift shows how general state variables can be modelled by behaviours, and thus shows that Fran can provide a foundation for concurrent, state-based but *declarative* animation of systems.

9 A glimpse under the bonnet

Up to this point we have used a high-level subset of the facilities provided by Fran. Behaviours which evolve have been defined using `stepper`, which turns an `Event a` – that is a timed sequence of values from type `a` – into a piecewise-constant `Behavior a`. We have also handled event occurrences in a simple-minded way, using the `-=>` operator to transform occurrences `(t,v)` of the event `ev` into occurrences `(t,c)` of the event `ev -=> c`.

How in general can the occurrence of an event cause a change in behaviour; how in general are events handled? In this expository section we explain how the Fran system defines the two operations `handleE` and `untilB` which provide a primitive interface to behaviours and events, and we see as examples how to define `stepper` and `ev -=> c` from these primitives.[10]

9.1 Handling `Events`

Recall from Section 4 that an `Event a` is a sequence of timed occurrences, each of which has associated with it a value of type `a`. In general, three values characterise an event occurrence:

[10] This section is included for completeness of exposition, but obviously owes much to (Peterson *et al.*, 1997).

- the time of the occurrence;
- the value of the occurrence, and
- the remainder of the `Event`, after removing the occurrences up to and including the occurrence in question.

To handle an event occurrence, we transform these three values to a value, of type `b`, say. This is accomplished by a function of type

```
Time -> a -> Event a -> b
```

and we can apply such a function to each occurrence in a stream of occurrences to give a stream of occurrences of type `b`, that is an `Event b`. That is the effect of the general event handler,

```
handleE :: Event a -> (Time -> a -> Event a -> b) -> Event b
```

As a special case of `handleE` in which the handler function is constant Fran defines `-=>` thus:

```
(-=>) :: Event a -> b -> Event b
ev -=> c = ev `handleE` (\_ _ _ -> c)
```

Also of interest is the `Event`-equivalent of `map`,

```
(==>) :: Event a -> (a -> b) -> Event b
ev ==> f = ev `handleE` (\_ v _ -> f v)
```

which when applied to the timed stream `[(t0,x0),(t1,x1),...,(tn,xn),...` and the function `f` gives the result `[(t0,f x0),(t1,f x1),...,(tn,f xn),...`.

9.2 Modifying `Behaviors`

How can one sequence behaviours, so that one follows another? A simple solution is provided by

```
seqB :: Behavior bv => bv -> Event () -> bv -> bv
```

so that `seqB bh1 ev bh2` behaves as `bh1` until the first occurrence of `ev`, and after that behaves as `bh2`. One can also lift this sequencing to operate over behaviours dependent upon a `User` argument to give

```
seqUB :: Behavior bv => (User -> bv) -> (User -> Event ()) ->
                        (User -> bv) -> (User -> bv)
```

and so forth. In fact, Fran provides a different primitive, `untilB`, which generalises these. It is, however, instructive, to see how `seqB` is defined from `untilB`. The function `untilB` has the type

```
Behavior bv => bv -> Event bv -> bv
```

and the effect of `` bh `untilB` ev `` is to behave as `bh` until the first occurrence of the event `ev`; after that the behaviour is whatever value is returned by that first event occurrence, which does indeed return a value of type `bv`. We can then see that

```
seqB bh1 ev bh2 = bh1 `untilB` (ev -=> bh2)
```

so that `seqB` separates the event and the continuation behaviour which are combined in the second argument to `untilB`. In a similar way, we define

```
seqUB ub1 uev ub2
  = \u -> ub1 u `untilB` (nextUser_ uev u ==> ub2)
```

where the Fran function `nextUser_` is used to age `Users` in the appropriate way. The effect of `nextUser_ uev u ==> ub2` is thus to pass the aged `User` to the `User`-lifted behaviour `ub2`, and so to continue the computation as required.

9.3 Using `handleE` and `untilB` to define the `stepper`

The Fran function `stepper` produces piecewise constant behaviour from a stream of values of type `a` and a starting value of that type.

```
stepper :: a -> Event a -> Behavior a
```

In what follows we show how the result `stepper start ev` is defined in a number of stages. Initially, the behaviour will be the constant behaviour with value `start`, that is

```
constantB start
```

If we think of `ev` as an infinite list, we then want recursively to call `stepper` on `(head ev)` and `(tail ev)`. We cannot do this directly, but we can *indirectly* using `handleE`. The Fran function

```
withRestE :: Event a -> Event (a,Event a)
withRestE ev = ev `handleE` (\_ head tail -> (head,tail))
```

returns the stream of ‘head,tail’ pairs from the event occurrences in `ev`. To apply `stepper` to each of these pairs, we ‘map’ it along `withRestE ev` using the `==>` operator, as follows

```
withRestE ev ==> uncurry stepper
```

where note that we have to `uncurry` the `stepper` function to accept its argument as a pair rather than ans two separate arguments. Putting all the parts together, we have

```
stepper start ev
  = (constantB start) `untilB` (withRestE ev ==> uncurry stepper)
```

This example shows how the exception handling mechanism of `handleE` gives a flexible way of dealing with `Events` – as timed streams of event occurrences – and also how the primitive `untilB` turns a stream of `Behaviors` into a single behaviour.

As we have seen, using the primitives `handleE` and `untilB` together with specializations of them and a number of other utility programs such as `nextUser_` it is possible to define a variety of powerful programs; in the section which follows we reflect on other aspects of using Fran.

10 Reflection

The solution presented in this paper is the result of a number of iterations which reflect a growing understanding of the capabilities of the Fran library as well as different approaches to the design of the simulation itself.

Terminating behaviours

For instance, in an earlier design the components of the solution were represented as terminating behaviours, which were sequenced together to form the overall simulation – a 'monadic' approach. Embedding Fran in a functional language with general-purpose capabilities makes modelling these terminating behaviours as pairs

```
( bv , Time -> User -> Event () )
```

a straightforward matter: the second components of these pairs are used to signal the termination of the behaviour in the first component. Such behaviours are sequenced using the analogue of `seqUB` from the previous section. This 'monadic' approach can be used in situations in which behaviours are built from components which are not piecewise continuous, for example.

Recursion

The solution we have presented here relies heavily on recursion. For example, `goUp` depends upon `pending` and `pending` depends upon `upPending` which in turn depends upon `goUp`. The implementation of Fran uses lazy evaluation heavily, so the recursive definition of structures is possible (we have done it here), but some recursions can lead to non-termination, and others to the system locking up.

This reflects a general phenomenon for which evidence is apparent in earlier approaches to functional I/O (see, for example, (Thompson, 1990)) and which led to the definition of sets of (monadic) combinators to handle I/O in a more disciplined manner (Thompson, 1990; Hughes & Peyton Jones, 1999).

Verification

One of the attractions of declarative programs is that it is substantially easier to reason about how they behave. The Fran library supports a declarative approach to behaviours which evolve in time, and a temporal logic (Emerson, 1990; Zhou Chaochen, 1994) can be used to prove properties of systems defined in Fran. Work in this direction is reported elsewhere (Thompson, 1999).

11 Conclusion

In this paper we have shown that Fran can be used to give a simulation of a lift, and we have argued that it is well suited to the task both because of its declarative model of the system and also because it is embedded in a general-purpose functional programming language, namely Haskell. This allows extensions of the library to be constructed with relatively small effort, and also allows those extensions to use features of Haskell – such as higher-order functions, polymorphism and type classes (for overloading) – in an essential way.

Other applications of the Fran model are reported in (Cameron *et al.*, 1999) and a logical view of Fran is explored in (Thompson, 1999).

References

Cameron, Helen, King, Peter, & Thompson, Simon. (1999). *Reactive Multimedia Documents in a Functional Framework*. `http://www.cs.ukc.ac.uk/people/staff/sjt/Fran`.

Elliott, Conal. (1997). *Composing Reactive Animations*. Available from `http://www.research.microsoft.com/~conal/fran`.

Elliott, Conal. (1998). Functional implementations of continuous modeled animation. *Proceedings of the joint conference PLILP/ALP*. Springer-Verlag.

Elliott, Conal, & Hudak, Paul. (1997). Functional Reactive Animation. *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP97)*. ACM Press.

Elliott, Conal, *et al.* . (1994). TBAG: A high-level framework for interactive, animated 3D graphics applications. *Proceedings of SIGGRAPH '94*. ACM Press.

Emerson, E Allen. (1990). Temporal and Modal Logic. van Leeuwen, Jan (ed), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier/MIT Press.

ETL. (1998). *Challenges for Executable Temporal Logics*. Further details at `http://www.cs.waikato.ac.nz/~dsmith/CHALLENGES/`.

Glasgow Haskell Compiler. (1998). *The Glasgow Haskell Compiler*. Available from `http://www.dcs.glasgow.ac.uk/fp/software/ghc/`.

Henderson, Peter. (1982). Functional Geometry. *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*. ACM Press.

Hudak, Paul, *et al.* . (1996). Haskore music notation – An algebra of music. *Journal of Functional Programming*, **6**.

Hughes, John, & Peyton Jones, Simon (eds). (1999). *Report on the Programming Language Haskell 98*. `http://www.haskell.org/report/`.

Hugs. (1998). *The Hugs System*. Available from `http://www.haskell.org/hugs/`.

Peterson, John, Elliott, Conal, & Ling, Gary Shu. (1997). *Fran Users' Manual*. Available from `http://www.haskell.org/fran`.

Thompson, Simon. (1990). Interactive functional programs: a method and a formal semantics. *Pages 249–285 of:* Turner, D.A. (ed), *Research Topics in Functional Programming*. Addison-Wesley.

Thompson, Simon. (1999). *Verifying Fran Programs*. Available from `http://www.cs.ukc.ac.uk/people/staff/sjt/Fran`.

Wadler, Philip. (1995). Monads for functional programming. Jeuring, Johan, & Meijer, Erik (eds), *Advanced Functional Programming*. Springer-Verlag, Lecture Notes in Computer Science, 925.

Zhou Chaochen. (1994). Duration Calculi: An Overview. Bjørner, Dines, *et al.* (eds), *Formal Methods in Programming and their Applications*. Springer-Verlag, Lecture Notes in Computer Science, 735.

Appendix: The complete program for the two floor lift

```
liftSim :: User -> ImageB

liftSim u

  = moveXY xPos yPos image

    where

    image :: ImageB

    image = stretch 0.3 circle

    xPos,yPos,dy :: RealB

    xPos = constantB 0
    yPos = lower + atRate dy u
    dy   = stepper 0 setRate

    setRate :: Event Double

    setRate = stop      -=> 0
              .|.
              goUp      -=> upRate
              .|.
              goDown    -=> downRate

    atBottom, atTop, stopped, waitingBottom, waitingTop :: BoolB

    atBottom = yPos <=* lower
    atTop    = yPos >=* upper
    stopped  = (dy ==* 0)

    waitingBottom = atBottom &&* stopped
    waitingTop    = atTop    &&* stopped

    arriveBottom, arriveTop :: Event ()

    arriveBottom = predicate (atBottom &&* dy <* 0) u
    arriveTop    = predicate (atTop    &&* dy >* 0) u
```

```
upButton, downButton, eitherButton :: Event ()

upButton     = lbp u
downButton   = rbp u
eitherButton = upButton .|. downButton

upPending, downPending, pending :: Behavior Bool

pending     = upPending ||* downPending

upPending   = stepper False (setUp   .|. resetUp)
downPending = stepper False (setDown .|. resetDown)

setUp, setDown :: Event Bool

setUp   = upButton   -=> True
setDown = downButton -=> True

resetUp, resetDown :: Event Bool

resetUp   = goUp   -=> False
resetDown = goDown -=> False

goDown, goUp, stop :: Event ()

goDown     = arriveTop    `whenE` pending
             .|.
             eitherButton `whenE` waitingTop

goUp       = arriveBottom `whenE` pending
             .|.
             eitherButton `whenE` waitingBottom

stop       = (arriveTop .|. arriveBottom) `whenE` notB pending
```